

ActuateOne™

One Design
One Server
One User Experience

Accessing Data
using e.Report Designer Professional

Information in this document is subject to change without notice. Examples provided are fictitious. No part of this document may be reproduced or transmitted in any form, or by any means, electronic or mechanical, for any purpose, in whole or in part, without the express written permission of Actuate Corporation.

© 1995 - 2011 by Actuate Corporation. All rights reserved. Printed in the United States of America.

Contains information proprietary to:

Actuate Corporation, 2207 Bridgepointe Parkway, San Mateo, CA 94404

www.actuate.com

www.birt-exchange.com

The software described in this manual is provided by Actuate Corporation under an Actuate License agreement. The software may be used only in accordance with the terms of the agreement. Actuate software products are protected by U.S. and International patents and patents pending. For a current list of patents, please see <http://www.actuate.com/patents>.

Actuate Corporation trademarks and registered trademarks include:

Actuate, ActuateOne, the Actuate logo, Archived Data Analytics, BIRT, Collaborative Reporting Architecture, e.Analysis, e.Report, e.Reporting, e.Spreadsheet, Encyclopedia, Interactive Viewing, OnPerformance, Performancesoft, Performancesoft Track, Performancesoft Views, Report Encyclopedia, Reportlet, The people behind BIRT, X2BIRT, and XML reports.

Actuate products may contain third-party products or technologies. Third-party trademarks or registered trademarks of their respective owners, companies, or organizations include:

Adobe Systems Incorporated: Flash Player. Apache Software Foundation (www.apache.org): Axis, Axis2, Batik, Batik SVG library, Commons Command Line Interface (CLI), Commons Codec, Derby, Shindig, Struts, Tomcat, Xerces, Xerces2 Java Parser, and Xerces-C++ XML Parser. Bits Per Second, Ltd. and Graphics Server Technologies, L.P.: Graphics Server. Bruno Lowagie and Paulo Soares: iText, licensed under the Mozilla Public License (MPL). Castor (www.castor.org), ExoLab Project (www.exolab.org), and Intalio, Inc. (www.intalio.org): Castor. Codejock Software: Xtreme Toolkit Pro. DataDirect Technologies Corporation: DataDirect JDBC, DataDirect ODBC. Eclipse Foundation, Inc. (www.eclipse.org): Babel, Data Tools Platform (DTP) ODA, Eclipse SDK, Graphics Editor Framework (GEF), Eclipse Modeling Framework (EMF), and Eclipse Web Tools Platform (WTP), licensed under the Eclipse Public License (EPL). Jason Hsueh and Kenton Varda (code.google.com): Protocole Buffer. ImageMagick Studio LLC.: ImageMagick. InfoSoft Global (P) Ltd.: FusionCharts, FusionMaps, FusionWidgets, PowerCharts. Mark Adler and Jean-loup Gailly (www.zlib.net): zlib. Matt Ingenthron, Eric D. Lambert, and Dustin Sallings (code.google.com): Spymemcached, licensed under the MIT OSI License. International Components for Unicode (ICU): ICU library. KL Group, Inc.: XRT Graph, licensed under XRT for Motif Binary License Agreement. LEAD Technologies, Inc.: LEADTOOLS. Matt Inger (sourceforge.net): Ant-Contrib, licensed under Apache License V2.0, Apache Software License. Microsoft Corporation (Microsoft Developer Network): CompoundDocument Library. Mozilla: Mozilla XML Parser, licensed under the Mozilla Public License (MPL). MySQL Americas, Inc.: MySQL Connector. Netscape Communications Corporation, Inc.: Rhino, licensed under the Netscape Public License (NPL). OOPS Consultancy: XMLTask, licensed under the Apache License, Version 2.0. Oracle Corporation: Berkeley DB. PostgreSQL Global Development Group: pgAdmin, PostgreSQL, PostgreSQL JDBC driver. Rogue Wave Software, Inc.: Rogue Wave Library SourcePro Core, tools.h++. Sam Stephenson (prototype.conio.net): prototype.js, licensed under the MIT license. Sencha Inc.: Ext JS. Sun Microsystems, Inc.: JAXB, JDK, Jstl. ThimbleWare, Inc.: JMemcached, licensed under the Apache Public License (APL). World Wide Web Consortium (W3C)(MIT, ERCIM, Keio): Flute, JTIty, Simple API for CSS. XFree86 Project, Inc.: (www.xfree86.org): xvfb. Yuri Kanivets (code.google.com): Android Wheel gadget, licensed under the Apache Public License (APL). ZXing authors (code.google.com): ZXing, licensed under the Apache Public License (APL).

All other brand or product names are trademarks or registered trademarks of their respective owners, companies, or organizations.

Document No. 111021-2-130381 October 14, 2011

Contents

About Accessing Data using e.Report Designer Professional	xv
--	-----------

Part 1

Using Actuate e.Report Designer Professional data access technology

Chapter 1

Accessing a data source	3
About accessing data for reports	4
About data access terms and relationships	4
Connecting to the data source from a report design	6
Choosing where to place a connection component	7
Creating a connection component	7
Specifying the data to retrieve from a data source	9

Chapter 2

Migrating reports and reusing report components	11
About migrating reports and reusing report components	12
Using a connection configuration file	12
Selecting an existing connection configuration file	13
Understanding connection configuration files	14
Using a connection configuration file to access data sources	18
Choosing whether to place data connection and data source components in a library	18
Specifying a connection for use in designing a report	19
Specifying a connection for use in running a report	22
Specifying connections in the Runtime element for use when running the report	22
Using the ConfigKey property to set a different connection when a report runs	23
Choosing a connection or data source component from a connection configuration file	24

Chapter 3

Modifying data processing	27
About modifying data processing	28
Modifying handling of a connection component	28
About data adapters, data source components, and data filters	28
Combining data adapters to form a data stream	29
Instantiating data adapters	30
Customizing a data stream by modifying data adapter methods	31
Adding code for additional data stream tasks	31

Accessing data in non-sequential order	32
Using data filters to process rows	33
About data rows	34
Creating a computed field	36
Accessing other types of data sources	38
Sorting data	39
Sorting data within the data source	40
Avoiding sorting by the group section sort key	40
Specifying the sort order using the OrderBy property	41
Specifying the sort order for sorting internally	42
Filtering data	43
Working with data from multiple sources	45
Creating a merge filter to combine the rows of the data sources	46
Creating a union filter to process the data sources sequentially	50
 Chapter 4	
Displaying data rows	53
About displaying data rows	54
Modifying a report design to display data rows	55
Specifying the columns to display data rows	58
Changing the order of columns	58
Modifying, adding, and deleting columns	60
Modifying font attributes for displaying data rows	67
 Chapter 5	
Accessing an Actuate Information Object	71
About information objects	72
Working with an information object	73
Preparing to access information object data	73
Setting up the report design to access information objects	73
Modifying font attributes for information object query output in Actuate Query	77
 Chapter 6	
Accessing data in a comma-separated values (CSV) text file	79
Accessing data from a CSV text file	80
Creating a custom data source	80
Creating variables to identify which text file to use	81
Defining data row variables	83
Displaying text file data in a report design	84
Specifying processing of the data	84
Running and viewing the report	86

Part 2

Accessing data in a database or ODBC data source

Chapter 7

Connecting to a database or ODBC data source 89

About database and ODBC connections	90
Preparing to access data using a database or ODBC driver	90
Using an ODBC driver	90
Preparing to access data using an ODBC driver	90
Configuring an ODBC driver	91
Using a native database driver	93
Defining a database or ODBC connection	94
Creating a query data source	96
About AFC support for database and ODBC connections and queries	99

Chapter 8

Creating a database or ODBC query 101

About creating databases and ODBC queries	102
Creating a query	102
Creating a query graphically	103
Opening Query Editor and Database Browser	104
Using tables, views, and synonyms	106
Creating table joins	109
About the automatic generation of joins	109
Creating and deleting joins manually	110
Modifying the generated FROM clause for a query	112
Selecting and modifying columns and creating computed fields	113
Selecting columns from a table	113
Creating a computed field	115
Changing the order of columns or the data type of a column	116
Summarizing data from multiple rows	117
Adding conditions to a query	120
About how e.Report Designer Professional applies conditions	120
Using QBE to specify a condition	122
Putting a condition on row retrieval	122
Putting conditions on an aggregate row	124
Viewing the generated SQL SELECT statement	126
Creating a query by writing a SQL SELECT statement	128
Writing the SQL query	128
Preparing the query	130
Modifying the textual query	131
Accepting the textual query	131

Converting a graphical query to a textual query	131
Previewing the rows that a database or ODBC query returns	133
Changing the properties of a result column	135
Changing the data type of graphical query results	135
Changing the display name of query results	136
Changing the data type, display length, and reference name of textual query results	137
Specifying sorting	139
Specifying data source sorting using a graphical query	140
Specifying sorting for a textual query	141
Overriding sorting by the group section sort key	141

Chapter 9

Filtering data 143

About user-specified filtering in database and ODBC queries	144
Prompting the user to provide a specific value to filter results	145
Using a simple call to a static parameter to specify a condition	146
Using SQL to specify a condition or other query clause	147
Preparing a textual query to describe a static parameter	148
Specifying a parameter's property values	148
Filtering query results with user-supplied expressions	149
Filtering a graphical query using a user-supplied expression	150
Filtering a textual query using a user-supplied expression	150
Including an ad hoc parameter in a textual query	151
Preparing a textual query to describe an ad hoc parameter	151
Modifying the property values of the ad hoc parameter	151

Chapter 10

Maintaining a database or ODBC query 153

About maintaining database and ODBC queries	154
Timing and optimizing data source queries	154
Viewing the SQL SELECT statement that e.Report Designer Professional sends to the data source	154
Modifying the SQL code in a query to optimize the query	155
Updating a query when the data source changes	156
Sending SQL statements to change the data source	157
Updating a textual query when the data source changes	157
Updating a graphical query when the data source changes	158
Ensuring that a query connects to the correct data source	159
Updating the data dictionary during synchronization	161
Determining whether a graphical query and its data source are synchronized	162
Synchronizing a graphical query	163
Verifying that synchronization is complete	166

Chapter 11

Accessing data using a stored procedure 167

About accessing a database using a stored procedure	168
Preparing to use a stored procedure to access a database	169
Designing a report that uses a stored procedure	169
Selecting a stored procedure	171
Changing the name of a stored procedure component	173
Limiting which stored procedures are available for selection	174
Working with data from a stored procedure	175
Using parameters with stored procedures	176
Specifying whether parameters are input or output parameters	176
Working with a sample value for an input parameter	178
Setting a Sybase Stored Procedure parameter to an empty string	179
Ensuring that the stored procedure is synchronized with the database	180
Using Oracle stored procedures	180
Using Oracle data types	180
Accessing stored functions	181
Identifying stored procedures and stored functions	181
Processing additional cursors that a procedure or function returns	181
Understanding how e.Report Designer Professional works with a stored function	182
About the EMP table in the example	182
About the stored function	182
About the result columns and parameters	183
Supplying parameter values	185
Viewing the report results	186
Customizing access to handle specific databases or multiple result sets	186
Accessing a stored procedure	186
Mapping Actuate variable types and Visual Basic type codes	187
Working with an Oracle stored procedure	188
Working with a stored procedure's return value	188
Working with a stored procedure to return an ID	189
Accessing multiple result sets from Oracle stored procedures	190
Adding support in Actuate Basic methods	191
Using the CPointer type with another stored procedure function	192
Fetching the data row	193
Calling the Oracle stored procedure with result sets	193

Part 3

Using Actuate open data access technology

Chapter 12

Accessing a custom data source 199

About accessing additional types of data sources	200
--	-----

Accessing a custom data source using an ODA driver	200
Building a custom data stream component	202
Communicating with a custom data source builder	202
About data-stream definition file elements	203
DataStreamDefinition element	203
DesignSessionRequest element	203
DesignSessionResponse element	204
ResultSetDefinition element	204
Providing access to data during report generation	204
Providing configuration information about a custom data driver	206
Installing a custom data driver	207

Chapter 13

Configuration file XML schema reference 209

About the configuration file for a custom data source	210
Understanding the schema of odaconfig.xml	210
Using the elements of odaconfig.xml	215
AlternativeOdaDataTypes	218
ArrayOfString	218
Connection	218
DataSource	219
DataSources	220
DataTypeMapping	220
DataTypeMappings	221
DesignerSpecific	221
DesignerSpecificProperties	221
DesignerSpecificType	222
DesignTimeInterface	222
DriverLibraries	223
InterfaceType	224
LibrariesForOs	224
LibrariesForOsType	225
ListOfProperties	225
NativeDataTypeType	226
OdaScalarDataType	226
OpenDataAccessConfig	227
OSType	228
Properties	229
Property	229
PropertyType	230
RunTimeInterface	230
TraceLogging	231
VendorInfo	232

Chapter 14

Custom data driver XML reference 235

Communicating the structure of a custom data source	236
Data source builder reference	236
ArrayOfInputFieldDefinition	236
ArrayOfInputParameterDefinition	236
ArrayOfNameValuePair	237
ArrayOfOutputFieldDefinition	237
ArrayOfOutputParameterDefinition	238
ArrayOfProperty	238
ArrayOfResultColumn	238
ArrayOfString	239
AxisType	239
ColumnAxisInfo	239
ConnectionProperties	240
DataStreamDefinition	240
DesignSessionRequest	242
DesignSessionResponse	243
DynamicConditionReference	244
DynamicQuery	245
ExternalState	245
HorizontalAlignment	246
InputFieldDefinition	246
InputParameterDefinition	248
NameValuePair	251
OdaDataType	252
OdaScalarDataType	253
OutputFieldDefinition	254
OutputParameterDefinition	255
Property	255
ResponseState	256
ResultColumn	256
ResultSetDefinition	259
SessionEditMode	260
SessionLocale	260
StateInfo	262

Chapter 15

Custom data driver Java reference 263

About Java interfaces and Java classes for creating a custom data driver	264
Open data access Java reference	265
Class FileHandler	266
FileHandler.close	267

FileHandler.publish	267
Filter	268
Filter.isLoggable	268
Class Handler	269
Handler.close	270
Handler.flush	270
Handler.getFilter	270
Handler.getFormatter	270
Handler.getLevel	270
Handler.getLoggingErrorHandler	271
Handler.isLoggable	271
Handler.publish	271
Handler.reportError	271
Handler.setFilter	272
Handler.setFormatter	272
Handler.setLevel	272
Handler.setLoggingErrorHandler	272
IBlob	273
IBlob.getBinaryStream	273
IBlob.getBytes	273
IBlob.length	274
ICallStatement	275
ICallStatement.findOutParameter	276
ICallStatement.getBigDecimal	277
ICallStatement.getBlob	277
ICallStatement.getClob	278
ICallStatement.getDate	278
ICallStatement.getDouble	279
ICallStatement.getInt	279
ICallStatement.getMetaDataOf	280
ICallStatement.getResultSet	280
ICallStatement.getResultSetNames	281
ICallStatement.getRow	281
ICallStatement.getSortSpec	281
ICallStatement.getString	282
ICallStatement.getTime	282
ICallStatement.getTimestamp	283
ICallStatement.setNewRow	283
ICallStatement.setNewRowSet	284
ICallStatement.setSortSpec	285
ICallStatement.isNull	285
IClob	287
IClob.getCharacterStream	287

IClob.getSubString	287
IClob.length	288
ICConnection	289
ICConnection.close	289
ICConnection.commit	289
ICConnection.createStatement	290
ICConnection.getMetaData	290
ICConnection.isOpened	291
ICConnection.open	291
ICConnection.rollback	291
ICConnectionFactory	292
ICConnectionFactory.getConnection	292
ICConnectionFactory.setLogConfiguration	292
ICConnectionMetaData	294
ICConnectionMetaData.getConnection	295
ICConnectionMetaData.getDriverMajorVersion	295
ICConnectionMetaData.getDriverMinorVersion	295
ICConnectionMetaData.getDriverName	295
ICConnectionMetaData.getDriverVersion	295
ICConnectionMetaData.getMaxConnections	296
ICConnectionMetaData.getMaxStatements	296
ICConnectionMetaData.getOdaMajorVersion	296
ICConnectionMetaData.getOdaMinorVersion	296
IDataSourceMetaData	297
IDataSourceMetaData.getConnection	299
IDataSourceMetaData.getDataSourceMajorVersion	299
IDataSourceMetaData.getDataSourceMinorVersion	299
IDataSourceMetaData.getDataSourceObjects	299
IDataSourceMetaData.getDataSourceProductName	300
IDataSourceMetaData.getDataSourceProductVersion	300
IDataSourceMetaData.getSortMode	300
IDataSourceMetaData.getSQLStateType	301
IDataSourceMetaData.supportsInParameters	301
IDataSourceMetaData.supportsMultipleOpenResults	301
IDataSourceMetaData.supportsMultipleResultSets	302
IDataSourceMetaData.supportsNamedParameters	302
IDataSourceMetaData.supportsNamedResultSets	302
IDataSourceMetaData.supportsOutParameters	302
IParameterMetaData	303
IParameterMetaData.getParameterCount	304
IParameterMetaData.getParameterMode	304
IParameterMetaData.getParameterType	305
IParameterMetaData.getParameterTypeName	305

IPParameterMetaData.getPrecision	305
IPParameterMetaData.getScale	306
IPParameterMetaData.isNullable	306
IResultSet	307
IResultSet.close	308
IResultSet.findColumn	308
IResultSet.getBigDecimal	308
IResultSet.getBlob	309
IResultSet.getClob	310
IResultSet.getDate	310
IResultSet.getDouble	311
IResultSet.getInt	311
IResultSet.getMetaData	312
IResultSet.getRow	312
IResultSet.getString	312
IResultSet.getTime	313
IResultSet.getTimestamp	313
IResultSet.next	314
IResultSet.setMaxRows	314
IResultSet.wasNull	314
IResultSetMetaData	315
IResultSetMetaData.getColumnCount	316
IResultSetMetaData.getColumnDisplayLength	316
IResultSetMetaData.getColumnLabel	316
IResultSetMetaData.getColumnName	317
IResultSetMetaData.getColumnType	317
IResultSetMetaData.getColumnTypeName	317
IResultSetMetaData.getPrecision	317
IResultSetMetaData.getScale	318
IResultSetMetaData.isNullable	318
IRowSet	319
IRowSet.absolute	320
IRowSet.add	320
IRowSet.clear	320
IRowSet.isEmpty	320
IRowSet.previous	321
IRowSet.setBigDecimal	321
IRowSet.setDate	321
IRowSet.setDouble	322
IRowSet.setInt	323
IRowSet.setString	323
IRowSet.setTime	324
IRowSet.setTimestamp	325

IResultSet.size	325
IStatement	326
IStatement.clearInParameters	328
IStatement.close	328
IStatement.execute	328
IStatement.executeQuery	329
IStatement.findInParameter	329
IStatement.getMaxRows	329
IStatement.getMetaData	330
IStatement.getMoreResults	330
IStatement.getParameterMetaData	330
IStatement.getParameterType	331
IStatement.getResultSet	331
IStatement.getSortSpec	331
IStatement.prepare	332
IStatement.setBigDecimal	332
IStatement.setDate	332
IStatement.setDouble	333
IStatement.setInt	333
IStatement.setMaxRows	334
IStatement.setProperty	334
IStatement.setPropertyInfo	335
IStatement.setSortSpec	335
IStatement.setString	335
IStatement.setTime	336
IStatement.setTimestamp	337
Class Level	338
Level.equals	340
Level.getName	340
Level.hashCode	340
Level.intValue	340
Class LogFormatter	341
LogFormatter.format	341
Class Logger	343
Logger.config	344
Logger.fine	344
Logger.finer	344
Logger.finest	344
Logger.getHandler	345
Logger.getLevel	345
Logger.getName	345
Logger.info	345
Logger.isLoggable	345

Logger.log	346
Logger.setHandler	346
Logger.setLevel	346
Logger.severe	347
Logger.warning	347
Class LoggingErrorHandler	348
LoggingErrorHandler.error	349
Class LogManager	350
LogManager.createLogger	351
LogManager.getLogger	351
Class LogRecord	353
LogRecord.getLevel	353
LogRecord.getMessage	354
LogRecord.getMillis	354
LogRecord.getThrown	354
LogRecord.setLevel	354
LogRecord.setMessage	354
LogRecord.setMillis	355
LogRecord.setThrown	355
Class OdaException	356
External exceptions	356
Localizing exceptions	356
Unsupported operations	357
OdaException.getCause	358
OdaException.getErrorCode	358
OdaException.getNextException	359
OdaException.getSQLState	359
OdaException.initCause	359
OdaException.setNextException	359
Class SimpleFormatter	360
SimpleFormatter.format	360
Class SortSpec	361
SortSpec.addSortKey	362
SortSpec.getSortColumn	363
SortSpec.getSortColumns	363
SortSpec.getSortKeyCount	364
SortSpec.getSortMode	364
SortSpec.getSortOrder	364
SortSpec.toString	365
Class StreamHandler	366
StreamHandler.close	367
StreamHandler.finalize	367
StreamHandler.flush	367

StreamHandler.isLoggable 367

StreamHandler.publish 367

StreamHandler.setFormatter 368

StreamHandler.setOutputStream 368

Class StringSubstitutionUtil 369

StringSubstitutionUtil.getDelimitedStringCount 369

StringSubstitutionUtil.substituteByIndex 372

StringSubstitutionUtil.substituteByName 373

Index 377

About Accessing Data using e.Report Designer Professional

Accessing Data using e.Report Designer Professional provides information about using Actuate e.Report Designer Professional to access data for your reports. This manual explains the general concepts for accessing data and the procedures for accessing data from a variety of standard sources. It also provides information and a reference guide for creating your own data drivers to access additional data sources.

e.Report Designer Professional is supported on Windows 7, Windows Vista, and Windows XP platforms. The default installation location on Windows 7 platforms is \Program Files (x86)\Actuate11\eRDPro. The default installation location on Windows Vista and Windows XP platforms is \Program Files\Actuate11\eRDPro. In paths throughout this document, the default installation location is represented by <eRDPro_HOME>. Illustrations show the default installation path on Windows XP platforms.

Accessing Data using e.Report Designer Professional includes the following chapters:

- *About Accessing Data using e.Report Designer Professional.* This chapter provides an overview of this guide.
- *Part 1. Using Actuate e.Report Designer Professional data access technology.* This part describes the general concepts for accessing data and also describes how to migrate a report to a production data source and how to access data from a text file.
- *Chapter 1. Accessing a data source.* This chapter explains data access concepts, including data access terms and how to optimize data access tasks. It also provides information and procedures for sorting data, filtering data to limit the returned rows, and working with data from multiple sources.
- *Chapter 2. Migrating reports and reusing report components.* This chapter provides information and procedures for reusing data and other components using

configuration files. It also provides information and procedures for migrating a report from using development to production data connections without recompiling the report design.

- *Chapter 3. Modifying data processing.* This chapter provides information and procedures for customization of Actuate e.Report Designer Professional's default data access processing.
- *Chapter 4. Displaying data rows.* This chapter provides information and procedures for accessing the output of Actuate Query as data for your report.
- *Chapter 5. Accessing an Actuate Information Object.* This chapter provides information about accessing data from information objects created in Actuate Information Object Designer.
- *Chapter 6. Accessing data in a comma-separated values (CSV) text file.* This chapter provides information and procedures for accessing data from text files.
- *Part 2. Accessing data in a database or ODBC data source.* This part provides information and procedures for accessing data from a database or ODBC data source.
- *Chapter 7. Connecting to a database or ODBC data source.* This chapter provides information and procedures for connection to a database or ODBC data source.
- *Chapter 8. Creating a database or ODBC query.* This chapter provides information and procedures for creating a query to access data from a database or ODBC data source.
- *Chapter 9. Filtering data.* This chapter provides information and procedures for providing parameters to enable users to filter data from a database or ODBC data source.
- *Chapter 10. Maintaining a database or ODBC query.* This chapter provides information and procedures for tuning a query and keeping a query synchronized to a database or ODBC data source.
- *Chapter 11. Accessing data using a stored procedure.* This chapter provides information and procedures for accessing data from database stored procedures.
- *Part 3. Using Actuate open data access technology.* This part provides information, procedures, and reference guides for creating your own drivers to access additional types of data sources.
- *Chapter 12. Accessing a custom data source.* This chapter provides information and procedures for creating your own custom data driver, including a description of the ODA configuration file schema.
- *Chapter 13. Configuration file XML schema reference.* This chapter provides a description of the ODA configuration file schema.

- *Chapter 14. Custom data driver XML reference.* This chapter provides a reference guide for the XML elements to use in your custom data driver to describe the structure of your data source.
- *Chapter 15. Custom data driver Java reference.* This chapter provides a reference guide for the Java interfaces and Java classes required for creating a custom data driver.

Part One

**Using Actuate e.Report Designer
Professional data access technology**

Accessing a data source

This chapter contains the following topics:

- About accessing data for reports
- About data access terms and relationships
- Connecting to the data source from a report design
- Specifying the data to retrieve from a data source

About accessing data for reports

Designing a report involves the following tasks:

- Plan the report
- Start a new report design
- Access data
- Lay out the report
- Format the contents of the report
- Customize the page layout
- Preview and test the report

This guide focuses on accessing data from report designs using Actuate e.Report Designer Professional. For more information about other report development tasks, see *Developing Reports using e.Report Designer Professional*. Although you do not need an understanding of all of the issues that are discussed in *Developing Reports using e.Report Designer Professional*, consider reading Part 1, “Getting started,” to gain an understanding of the report design process before you use the information in this book.

This chapter defines data access terms and provides general information about creating components to access data. After familiarizing yourself with these general concepts, consider reading the chapters about reusing components, migrating reports to production environments, modifying the default data processing, and displaying data rows before formatting a report. Other chapters discuss how to access data specifically from each of the following types of data sources:

- Database and Open Database Connectivity (ODBC)
- Information object
- Comma-separated values (CSV) text file

You also can access XML data by using Actuate Information Object Designer to create an information object that accesses data from XML files. Information objects can access and combine information from databases and XML files.

About data access terms and relationships

Before a report can display data, it must obtain the data from a data source. Typically, the data source is a database. Other potential data sources include files, information objects, and so on. The computer that generates the report must have a local or network connection to the data source.

You obtain data from the data source using a data stream. Figure 1-1 shows how data flows from a data source, through a connection, to an Actuate report.

Table 1-1 describes the components that are involved in accessing data.

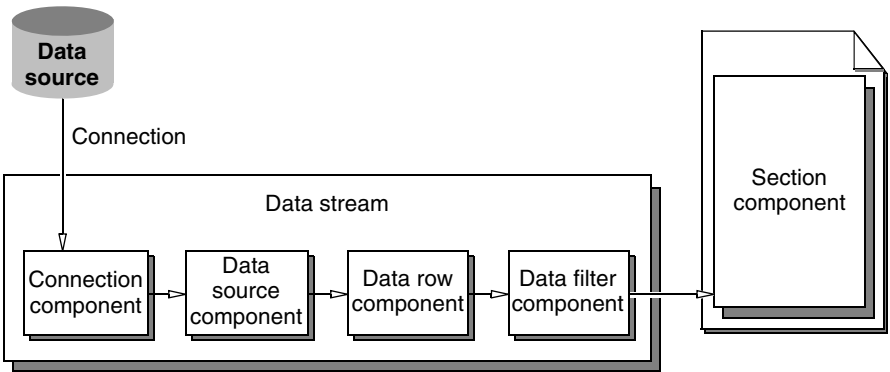


Figure 1-1 Data flow from data source to Actuate report

Table 1-1 Components of accessing data

Component	Description
Connection	Stores the information that the report needs to connect to the data source. A section can use a connection component in its own connection slot or a connection in the Connection slot of the data source component. If a data source component does not have its own connection, it uses the connection component that is defined above it. The connection component is also called a database connection component. You can use connection components to connect to databases and other types of data sources. Some types of data sources, such as text file, do not need a connection component. For example, e.Report Designer Professional opens a text file and processes its contents based on custom code and variables that identify the file and how to read the data.
Data filter	Optional. Performs additional processing to put the data in the form that the report requires. Each filter passes a data row to the next filter in the data stream, if another filter exists. If there are no filters, the data row from the data source component is passed directly to the report. Filters are used to omit rows, sort rows, compute new values, perform lookups, or combine data from several sources.
Data row	Consists of a set of variables that contain the data from the data source.

(continues)

Table 1-1 Components of accessing data (continued)

Component	Description
Data source	Selects data from the data source and puts the data in data rows. For most types of data sources, the data source component uses a connection component to connect to the data source. For example, a data source component that accesses data from a database uses the connection component to the database. A text file is an example of a data source that does not need a connection component.
Section	Receives data rows and displays them.

In addition to these components, the terms in Table 1-2 refer to components or groups of components:

Table 1-2 Component terminology

Term	Description
Data adapter	A data source component or a data filter component. Data adapters collect, process, and deliver data rows to report sections.
Data component	All types of data-related components that are available on Toolbox—Data. A data component can be a data connection component, data source component, data row component, or data filter component.
Data stream	A group of components, including data adapters, that selects data from the data source and places the data in the data row structure that the report requires. A data stream contains a data source component, a data row component, and optionally one or more data filter components. The data stream can require a connection component, depending on the type of data source. If the data stream requires a connection component, the data stream can use the connection component that is defined within the data stream, if one exists. If the data stream does not contain a connection component, the data stream can use the closest connection component above it. For example, if a report section contains the data stream, the data stream can use the report section's connection component. For information about sections, see <i>Developing Reports using e.Report Designer Professional</i>.

Connecting to the data source from a report design

A connection is a communication link from a computer to a data source. A connection component establishes and maintains the connection between a data source and a report. e.Report Designer Professional uses the connection to:

- Get lists of tables and columns from which you choose when you design a report.

- Provide access to data for a report during report generation.

You can create report sections without a connection component in the following situations:

- If the report section shares a connection with an outer report section. For more information about sharing a connection, see *Developing Reports using e.Report Designer Professional*.
- If the report section does not use data or gets data from a type of data source that does not use a connection component. You can handle data from additional types of data sources by creating a custom data stream. For example, the method of data access from CSV files in e.Report Designer Professional uses a custom data stream.

To deploy report object executable (.rox) files using a connection, you must also define the connection on Actuate iServer. For more information about defining a database connection on Actuate iServer, see *Configuring BIRT iServer*.

Choosing where to place a connection component

The data source component uses the connection component that is closest to it in the report structure. If a data source component does not contain its own connection, e.Report Designer Professional searches upward in the report structure until it finds a connection. A connection in a data source takes precedence over a connection in a section. A section of a report design can have its own connection component or inherit the connection component of an enclosing section. Sequential, conditional, and parallel sections access data indirectly through the enclosing section.

One way to work with a database connection is to place the connection in the data source component. Use this technique when a report uses a single data source component or if each data source component in the report requires a different connection. If you publish the data source component, the connection and data source components are published together.

Place a connection component in a report section or group section if multiple data source components can or must share the connection. For example, if you have two report sections that both require the same data source, you can create a report section that encloses both of them and specify the connection only in the enclosing report section. This approach can increase performance, because the data source only opens and closes one connection. You also can use this approach if your data source restricts the number of concurrent connections.

Creating a connection component

The connection component stores data source-specific information that is needed to access the data source server, such as server name, user name, and password. Each of these items is a property of the connection component. For example, the

user name is set in the UserName property. After you create a connection component, you can delete or modify it like any other component. For more information about components and properties, see *Developing Reports using e.Report Designer Professional*.

Actuate e.Report Designer Professional provides connection components for DB2, ODBC, Oracle, and information object connections. For some types of data sources, such as Oracle, you must install a database client or other software on your system so that your computer can access the data source. For more information about creating connections for a specific type of data source, see the one or more chapters specifically about that data source.

How to create a connection component

- 1 From the toolbox, select Data, as shown in Figure 1-2.

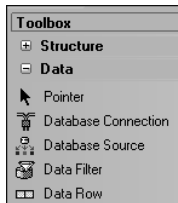


Figure 1-2 The data tools

- 2 Drag a database connection component from Toolbox—Data and drop it in a Connection slot, in Report Structure, as shown in Figure 1-3.

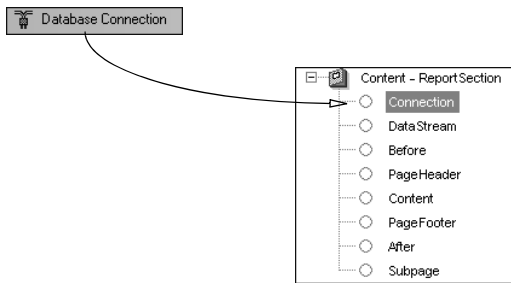


Figure 1-3 Adding a connection component

- 3 In Select Component, select the type of connection to create, as shown in Figure 1-4, then choose OK.

If you have already chosen a data source component, only connection types that are compatible with that data source component appear in this list.

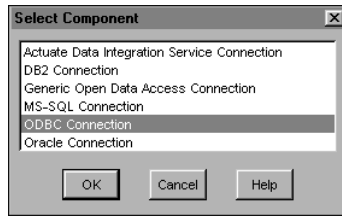


Figure 1-4 Selecting the type of connection

Component Properties appears, displaying the properties that are available for the selected type of connection. Figure 1-5 shows the properties that are available for an ODBC connection.

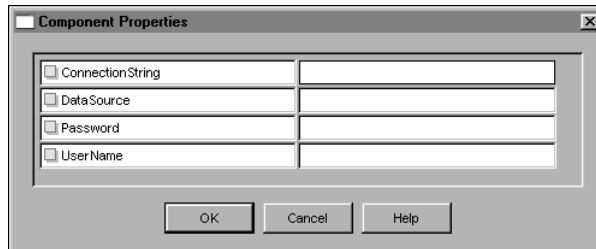


Figure 1-5 Properties for an ODBC connection

- 4 Type the connection properties that your data source server requires. For some properties, you can choose a value from a drop-down list.

If you do not specify the required connection properties for the connection component, Database Login appears and requests the connection properties when e.Report Designer Professional attempts to connect to the data source.

Specifying the data to retrieve from a data source

To specify the data that you want to retrieve from the data source, use a data source component. Actuate e.Report Designer Professional provides data source components for use with standard types of data sources, such as DB2, ODBC, Oracle, and information objects. For each of these types of data sources, e.Report Designer Professional provides an editor or wizard to specify the data to retrieve from the data source. You can then create a report design that uses some or all of the fields in the retrieved data. For better performance, limit the fields that are obtained by your data source component to just those fields that your report design requires. If you have data or connections that your organization needs to access frequently, consider using libraries and configuration files to facilitate reusability.

Migrating reports and reusing report components

This chapter contains the following topics:

- About migrating reports and reusing report components
- Selecting an existing connection configuration file
- Understanding connection configuration files
- Using a connection configuration file to access data sources
- Choosing a connection or data source component from a connection configuration file

About migrating reports and reusing report components

A report design can have an associated connection configuration file that e.Report Designer Professional loads when opening a report design. The file is an XML document that provides access to data connections, libraries, and report templates for use in report designs. You can standardize on a single e.Report Designer Professional connection configuration file for all projects or design a connection configuration file for each set of reports, such as the set of all financial reports for a company.

Connection configuration files enable migration of reports to a production environment. You can use a connection configuration file to specify which data connections to use in the design environment. You can use the same data connections or specify different connections for use when Actuate iServer runs the report. By specifying both design-time and run-time connections, you do not have to change the report design when migrating a report to a production environment.

Connection configuration files also support reuse of report components in multiple report designs, providing a mechanism for centralized control of the structure and format of report components. You can use e.Report Designer Professional connection configuration files to manage and set up easy access to libraries, report templates, and data components. You can place a library in a report design without using a connection configuration file but using a connection configuration file provides quick access to structural, data, and visual components for report building. For example, if all your reports use the same libraries, you can specify that e.Report Designer Professional automatically include these libraries in every report that you create. Alternately, you can create different connection configuration files for different purposes. For example, you can create separate connection configuration files for the Sales and Finance departments. Each connection configuration file can provide easy access to department-specific data streams, report templates, and libraries.

Using a connection configuration file

A connection configuration file is an XML file that you use to set data source connection settings that you want to use when a report is run on an Actuate iServer system. In e.Report Designer Professional, you can also use the file to configure other settings. The settings that you include in a connection configuration file can override the settings in a report. For example, you can develop a report using one data source, such as a test database. Then, you can create a connection configuration file to run the report with a different data source, such as the equivalent production database. To change or add connection

information for a spreadsheet report, you create or modify an existing connection configuration file with the connection information.

You can use the same connection configuration file for every spreadsheet report that you run on Actuate iServer. e.Spreadsheet Designer, e.Report Designer Professional, and BIRT all use the same connection configuration file. Each entry in the file is specifically for a particular product.

Actuate iServer expects the file to be in UTF8 encoding, which allows a variety of special characters. You can also use a file with only ASCII characters.

There is no default location for the connection configuration file. To use a connection configuration file, you create the file and then specify its location using the ConnConfigFile parameter in Actuate Management Console. Set the ConnConfigFile parameter to the absolute path and name of your connection configuration file. If you have a cluster of iServers, each iServer in the cluster needs to have access to the file. The path can be a local absolute path on each machine and must be specified for each iServer in the server configuration. If you use a single copy of the file for a cluster, put the file in a shared location and then specify the path to that shared location for all iServers in the cluster.

When you run a report, Actuate iServer uses the data source connection information in the connection configuration file specified in the ConnConfigFile parameter, if the file exists and the name of the data source is listed in the file. If the file does not exist or does not list the data source, then Actuate iServer uses the connection information from the definition of the data source in the spreadsheet report.

Selecting an existing connection configuration file

To help new report users get started quickly, e.Report Designer Professional's default report development environment includes a sample connection configuration file. Before you begin developing production reports, you should specify a connection configuration file appropriate for your development environment. If you do not replace or disable the default connection configuration file, all report designs that you create include a few Actuate-defined resources, such as a default connection component and sample libraries.

If your organization has created one or more connection configuration files, a report developer can select the appropriate connection configuration file by choosing Tools>Options>General and navigating to the file, as shown in Figure 2-1.

When you choose Refresh on Options—General, e.Report Designer Professional reads the connection configuration file. If there are errors in the file, Output displays error messages, as shown in Figure 2-2.

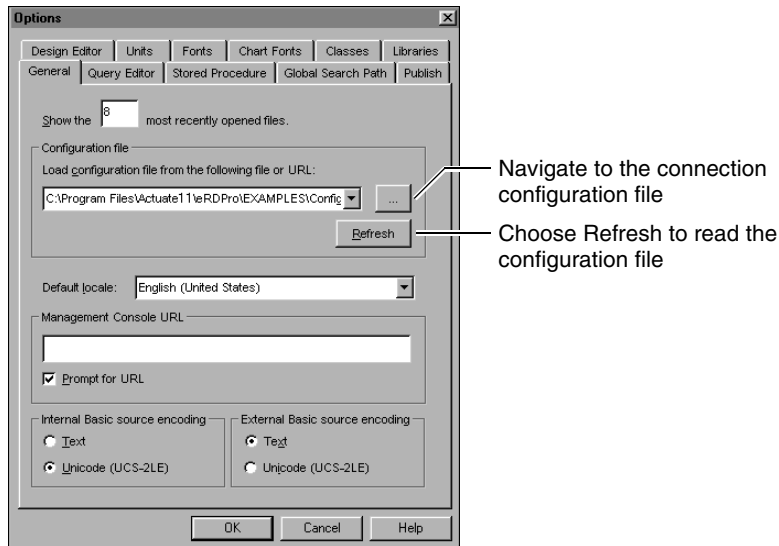


Figure 2-1 Navigating to the connection configuration file

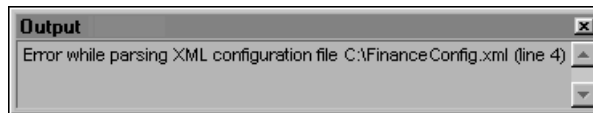


Figure 2-2 Error message display

If there are no errors in the file, e.Report Designer Professional invokes this connection configuration file when opening this report design or running the report.

You do not need to specify a connection configuration file to use e.Report Designer Professional. If you prefer to start your report designs with a blank slate, you can navigate to Options—General and delete the path and file name in Configuration File.

Understanding connection configuration files

An e.Report Designer Professional connection configuration file can contain the following types of information:

- The path along which to search for report design components
- The database connections to use for a report design
- The database connections to use for running a report on Actuate iServer
- The component and function libraries available to a report design

- The component and function libraries to include in a report design
- Templates on which to base report designs
- The path to other connection configuration files that provide access to additional resources

You can use any text editor to create a connection configuration file, as long as the resulting file contains only the text you type. The connection configuration file is an XML document composed of declarations, elements, comments, and processing instructions. The best way to create this document is to make a copy of the sample connection configuration file and replace the contents of the file with information specific to your development environment. The sample connection configuration file has the following name and location under the Actuate home directory:

```
\eRDPro\Examples\ConfigurationFile\sample_configuration_file.xml
```

A connection configuration file consists of the root element `<config>` and one or more of the top-level elements, listed in any order. The `<config>` root element is required. The format of the file appears as follows:

```
<config>
  <top-level element 1>...</top-level element 1>
  ...
  <top-level element n>...</top-level element n>
</config>
```

The top-level elements are described in Table 2-1.

Table 2-1 Connection configuration file top-level elements

Top-level element	Description
Design	Indicates which libraries, data connections, and template to make available to the report design. You can use only one Design element, which can specify multiple libraries, connections, and templates.
Include	Optional element that specifies that another connection configuration file's contents also should be read and used. You can use more than one Include element.
Runtime	Optional element that specifies the data connections to use when Actuate iServer runs the report. You can use only one Runtime element, which can specify multiple data connections.
SearchPath	Optional element that lists the location of the libraries to use with this connection configuration file. You can only use one SearchPath element, which can list the location of multiple library locations.

Your Design element can specify as many libraries, templates, and connections as you need by using the second-level elements listed in Table 2-2.

Table 2-2 Second-level elements in a Design element

Element within the Design element	Description
Connection	Optional element that specifies data connections to make available to a report design.
Library	Optional element that identifies the libraries to make available to a report design. For information about using libraries and the syntax for using the Library element in a connection configuration file, see <i>Developing Reports using e.Report Designer Professional</i> .
Template	Optional element that identifies a template to make available to a report design. For information about the syntax for using the Template element in a connection configuration file, see <i>Developing Reports using e.Report Designer Professional</i> .

For example, all of the top-level and Design elements are used in the following connection configuration file:

```
<Config>
  <SearchPath>
    <Location>\DesignResources</Location>
  </SearchPath>

  <Include>
    C:\Includes\StocksConfig.xml
  </Include>

  <Design>
    <Library
      Autoinclude="true"
      Alias="Financial Reports">
      C:\LibraryFiles\Finance.rol
    </Library>

    <Connection Type="ODBCConnection"
      Alias="Financial reporting resources"
      DefinedIn="C:\LibraryFiles\Stocks.rol"
      Description="For quarterly and weekly financial reports."
      IsDefault="True">
      <Property PropName="DataSource">
        ChartExamples
      </Property>
    </Connection>
```

```

<Template
  Alias="Finance Reports Template"
  Description="Template for Finance Department use.">
    C:\Includes\FinanceTemplate.rod
  </Template>
</Design>

<Runtime>
  <ConnectOptions
    Type="ODBCConnection">
      <Property PropName="DataSource">
        sfdata
      </Property>
    </ConnectOptions>
  </Runtime>
</Config>

```

Any new blank report that uses this connection configuration file includes three libraries and a reference to the default connection, as shown in Figure 2-3.

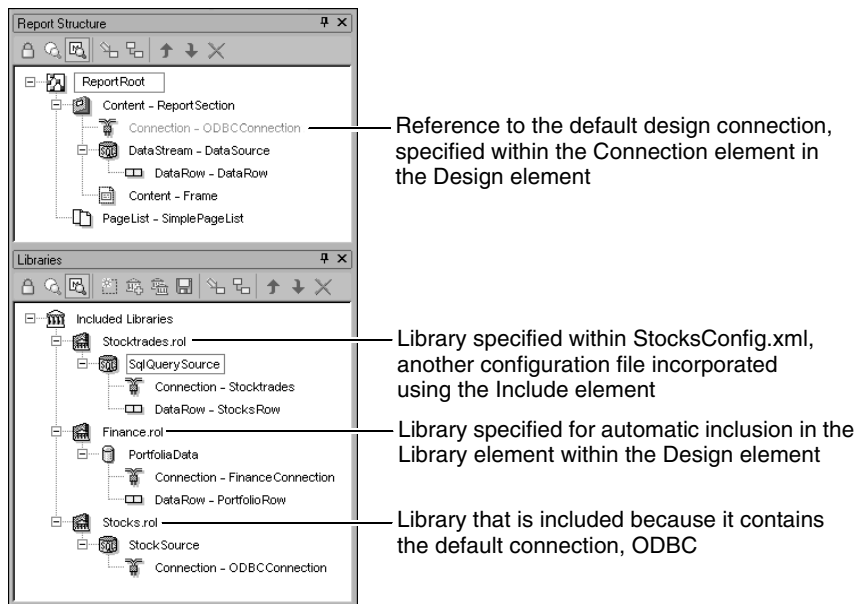


Figure 2-3 Blank report using the configuration file

Including another connection configuration file makes additional resources available to a report design. The Include element specifies other connection configuration files to include in the design. To include multiple connection configuration files, create an Include element for each file, as shown in the following example:

```
<Include>
  C:\Standard\APayableConfig.xml
</Include>

<Include>
  C:\Configuration\AReceivableConfig.xml
</Include>
```

Within each Include element, provide an absolute path or a path relative to the directory that contains the connection configuration file. Alternatively, you can provide an absolute or relative URL.

Using a connection configuration file to access data sources

A report can use one or more connections to data sources of various types. e.Report Designer Professional uses a connection to a data source when:

- A report designer creates or edits a query.
- A report designer runs a report from e.Report Designer Professional.
- A user runs a report on Actuate iServer.

Although you can define data connection and data source components for each report design or manually include a library containing these components, consider using a connection configuration file. If more than one report design uses the same data connection or data source components, using a connection configuration file can simplify the access and reuse of these components.

You also can use a connection configuration file to specify the use of one connection when designing a report and another connection of the same type when running the report. This capability enables migration from a development database to a production system without changing the report design files.

Choosing whether to place data connection and data source components in a library

Although you can define a data connection in the connection configuration file directly, consider including a data connection in a library as part of your component reuse strategy. If a connection is included in a library, you can use the connection configuration file to provide report designs access to both the library and the data connection.

If you use a library, you can publish connection components separately or as part of a data source component. Typically, connection components are published separately for greater reusability. You can improve reusability of a published data source component by specifying fields in the tables that you expect report designs

to need. To improve performance when you include that component in a report design, a report developer can modify it to retrieve only the fields that the report design requires. For information about creating libraries and the syntax for using the connection configuration file's Library element, see *Developing Reports using e.Report Designer Professional*.

Specifying a connection for use in designing a report

Use the Connection element in the connection configuration file's top-level Design element to specify one or more connections for a report design. Use a separate Connection element for each connection. If you do not specify a corresponding connection in the connection configuration file's Runtime element, then Actuate iServer uses the connection defined in the Connection element to run the report.

You can specify a connection using the parts of the Connection element, as shown in Table 2-3.

Table 2-3 Connection element parts

Part within the Connection element	Description
Type attribute	Specifies whether the report connects to an ODBC or other type of data source. You can use class name, such as AcODBCConnection, or you can use another descriptive term, such as ODBC or TradesDB. You can specify multiple connections of the same type. If a connection is in a library, the string you use for Type must match the string that appears in Report Structure for the connection in the library. Typically, string mismatches result from using spaces inconsistently.
Alias attribute	Optional element that defines the name of the component that creates the connection. The Design element can contain multiple entries for the same connection type, such as ODBC, but each entry must have a different alias. The alias appears in the Name column on Report Wizard—Choose Database.
Description attribute	Optional element that provides a brief explanation of the connection's purpose. The text appears in Description on Report Wizard—Choose Database.
DefinedIn attribute	Optional element that indicates a library that contains the connection. The path you specify in DefinedIn is relative to the directory that contains the connection configuration file.

(continues)

Table 2-3 Connection element parts (continued)

Part within the Connection element	Description
DefinedIn attribute (continued)	If the library that contains this connection is not in either the SearchPath element or the Library element, e.Report Designer Professional uses the DefinedIn path to include the library in the report design. If the library is not in the SearchPath, Library, or DefinedIn element, you must include the required library when you build a report that uses the connection.
IsDefault attribute	Optional element that indicates whether the connection is the default connection for blank report designs. If you set IsDefault to True for a connection, then a newly created blank report has that connection by default. If you set more than one default connection, the last connection in the connection configuration file becomes the default connection.
Property element	Defines one or more properties of the data source. The properties vary according to the type of data source. For example, an ODBC connection requires a DataSource property. For a list of the properties of each type of connection, see AcDBConnection, AcDB2Connection, AcMSSQLConnection, AcODACConnection, AcODBCCConnection, and AcOracleConnection in <i>Programming with Actuate Foundation Classes</i> .

The following code example shows how to specify an ODBC connection that is part of a library:

```
<Design>
  <Connection
    Type="ODBCConnection"
    Alias="Financial reporting resources"
    DefinedIn="C:\LibraryFiles\Stocks.rol"
    Description="For quarterly and weekly financial reports."
    IsDefault="true">
    <Property PropName="DataSource">
      stocktrades
    </Property>
  </Connection>
</Design>
```

You can include a connection without reference to a library by omitting DefinedIn, as shown in the following example:


```

<Design>
  <Connection
    Type="ODBCConnection"
    Alias="Customer Service database"
    Description="For quarterly and weekly service level reports">
    <Property PropName="DataSource">
      custserv
    </Property>
  </Connection>
</Design>

```

To externalize an AcAisConnection, you must specify the path to the library where AcAisConnection is defined. The library path depends on the installation and is specified in the DefinedIn attribute of the connection element. Set the Autoinclude attribute for the Library element to true. AcAisConnection is defined in the library AcAISDataSource.rol. On Windows XP platforms, the default installation path is C:\Program Files\Actuate11\oda\ais\AcAISDataSource.rol.

This example specifies an AIS connection to an iServer hosted on viper at port 8000:

```

<Design>
  <Library
    Autoinclude="true"
    Alias="My Connections">
      C:\Program Files\Actuate11\oda\ais\AcAISDataSource.rol
    </Library>
  <Connection
    Type="AcAisConnection"
    Alias="AIS_viper"
    DefinedIn="C:\Program Files\Actuate11\oda\ais\AcAISDataSource.rol"
    Description="Connects to AIS on viper"
    IsDefault="False">
    <Property PropName="ServerUri">
      http://viper:8000/
    </Property>
    <Property PropName="Volume">
      viper
    </Property>
    <Property PropName="UserName">
      administrator
    </Property>
    <Property
      PropName="Password" />
    </Connection>
</Design>

```

Specifying a connection for use in running a report

You can use one set of connections while designing a report and a different set when Actuate iServer runs the report. If you do not set up a different set of connections for running the report, then Actuate iServer uses the design connections. This is useful for migrating a report design from a development environment to a production environment without recompiling the report design.

You must use the same type of data source for report generation as for report design.

To specify a different connection for Actuate iServer to use in running a report, you must specify the connection in the connection configuration file's Runtime element and use the data connection's ConfigKey property to identify that runtime connection. The report design's connection's ConfigKey property identifies which connection specified in the Runtime element should be used by Actuate iServer to run the report. If you do perform these steps, then Actuate iServer uses the design connection to run the report.

To run the report on Actuate iServer using a connection configuration file, you also must set a path to the connection configuration file using Actuate iServer's Connection configuration file for connections parameter, ConnConfigFile. When you set this parameter, Actuate iServer uses the connection configuration file to run reports that require the file. If you do not set this parameter, a database error occurs for reports that require the connection configuration file. For more information about the connection configuration file for Connections parameter, see *Configuring BIRT iServer*.

Specifying connections in the Runtime element for use when running the report

The Runtime element of a connection configuration file specifies information about the connections for Actuate iServer to use for report generation. Use this element when you want to use different connections when designing the report than you use when Actuate iServer runs the report. A connection configuration file has one Runtime element, which can specify multiple connections. Within the Runtime element, you can use one or more ConnectOptions elements. Each ConnectOptions element specifies information needed for the connection, using the parts of the ConnectOptions element listed in Table 2-4.

Table 2-4 Parts of the ConnectOptions element

Part within the ConnectOptions element	Description
Type attribute	Specifies what type of connection the report uses. You can use appropriate class name, such as AcODBCConnection, or you can use another descriptive term, such as ODBC or TradesDB.

Table 2-4 Parts of the ConnectOptions element

Part within the ConnectOptions element	Description
Type attribute (<i>continued</i>)	If a connection is in a library, the string you use for Type must match the string that defines the connection in the library. Typically, string mismatches result from using spaces inconsistently.
Property element	Defines one or more properties of the data source. The properties vary according to the type of data source. For example, an ODBC connection requires a DataSource property. For a list of the properties of each type of connection, see AcDBCConnection, AcDB2Connection, AcMSSQLConnection, AcODACConnection, AcODBCCConnection, and AcOracleConnection in <i>Programming with Actuate Foundation Classes</i> .

In the connection configuration file's Runtime element, use a ConnectOptions element for each connection, as shown in the following example:

```
<Runtime>
  <ConnectOptions
    Type="ODBCConnection">
      <Property PropName="DataSource">
        finapps
      </Property>
    </ConnectOptions>
  </Runtime>
```

Using the ConfigKey property to set a different connection when a report runs

By setting a connection component's ConfigKey property in e.Report Designer Professional, you can preset the use of a different connection when Actuate iServer runs the report. By using ConfigKey, you can migrate a report design from a development environment to a production environment without recompiling the code.

Set the ConfigKey property of the report design's connection component to match the Type property of a connection in the Runtime element. Be sure that you use the same type of connection for designing the report and running the report. For example, if the report design's connection is an ODBC connection, be sure to specify another ODBC data source in ConfigKey. Do not change the DataSource property for the connection. The ConnectOptions element of the Runtime element specifies the data source. Figure 2-4 shows a ConnectOptions element and a ConfigKey element that references it.

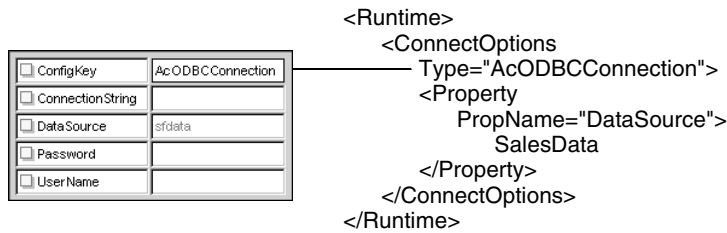


Figure 2-4 A ConnectOptions element and a corresponding ConfigKey property

Choosing a connection or data source component from a connection configuration file

If you specify a library in a connection configuration file that contains a data source component, a report developer can pick the data source component from a list of predefined data sources in the report wizard. If you specify a connection component in a connection configuration file, a report developer can pick the connection from a list of predefined databases and other types of data sources. The list of connections contains connection components that are specified directly or as part of a data source component. If you modify a data source that a component in a library references, the data source component is subclassed from the component in the library.

How to use a connection component that is specified in a connection configuration file

- 1 In Report Structure, as shown in Figure 2-5, select a report section, connection, or data source slot.

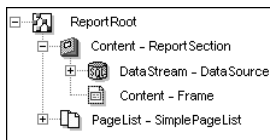


Figure 2-5 ReportStructure

- 2 Choose Tools→Database Connection. On Database Connection, select Choose from a list of predefined databases, as shown in Figure 2-6, then choose Next.

Database Connection—Choose Database lists the connections that are specified through the connection configuration file, as shown in Figure 2-7.

Select a connection, then choose Next. Connection properties that are relevant for the type of connection that you selected appear in Database Connection—Properties. For example, Figure 2-8 shows the connection properties for an ODBC data source.

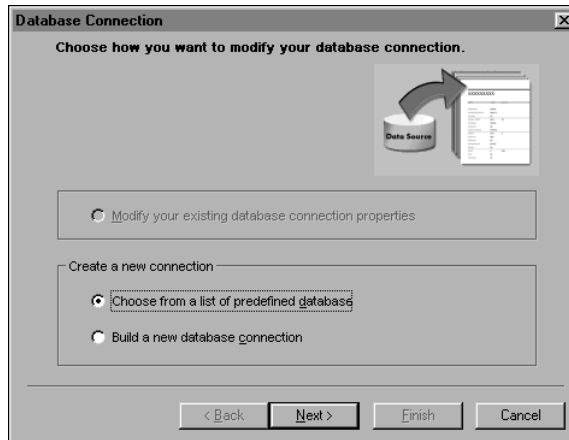


Figure 2-6 Choosing to use a list of predefined databases

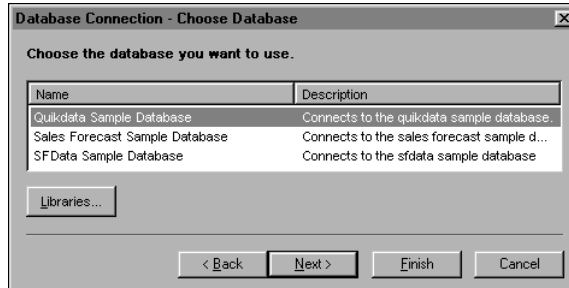


Figure 2-7 Connections listed in the connection configuration file

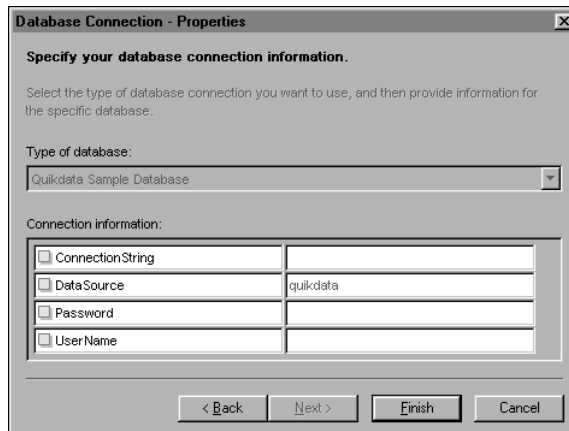


Figure 2-8 Connection properties for an ODBC data source

- 3 If necessary, modify the connection information by typing a value or choosing from a drop-down list. After completing modifications to connection information, choose Finish. Report Structure displays the selected connection component in the Connection slot, as shown in Figure 2-9.

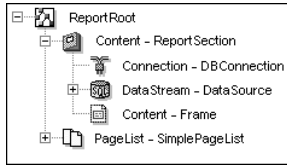


Figure 2-9 Report Structure with a connection component

Modifying data processing

This chapter contains the following topics:

- About modifying data processing
- Modifying handling of a connection component
- About data adapters, data source components, and data filters
- Sorting data
- Filtering data
- Working with data from multiple sources

About modifying data processing

Typically, e.Report Designer Professional's default functionality is sufficient for a report developer's data access needs. For information about this default functionality, see Chapter 1, "Accessing a data source."

e.Report Designer Professional also supports customization of its default data access functionality. This chapter provides additional information about the data components and describes techniques to modify standard data processing. This chapter assumes an understanding of data access terms and relationships, as described in "About data access terms and relationships" in Chapter 1, "Accessing a data source."

e.Report Designer Professional is an object-oriented program using classes written in Actuate Basic. Each report design component is a representation of a class. You can supplement or override the default functionality by using object-oriented programming to modify these classes. These classes reside in the Actuate Foundation Class library and some of them are represented by icons in e.Report Designer Professional.

Modifying handling of a connection component

Typically you only set the properties for a connection component and determine where to place the component. For information about placing a component to improve performance or support reuse, see "About data access terms and relationships" in Chapter 1, "Accessing a data source."

If desired, you can also customize the methods for the connection component. If you placed the connection in a data stream, the data adapter instantiates the connection by calling `NewConnection()` and opens it by calling `OpenConnection()`. You can override `NewConnection()` to customize the selection of a connection. You can override `OpenConnection()` to customize the connection before opening it.

If you placed the connection in a section, the section's `Start()` method instantiates the connection and passes it to the data source. If desired, you can customize the `Start()` method.

About data adapters, data source components, and data filters

A data adapter can be a data source component or a data filter component. A data source component retrieves data from an input source, such as a database and

creates data rows. A data filter component sorts, filters, or performs other computations on a data row. Data sources, data filters, and data rows are transient objects. The Factory deletes them after they retrieve, process, and deliver all data rows to a report.

Combining data adapters to form a data stream

A data stream has the following characteristics:

- A data stream must have at least one data source. It can have multiple data sources. If a report uses data from multiple input sources, there must be a data source for each input source.
- A data stream can have any number of data filters.

Figure 3-1 shows a data stream configuration that retrieves data from a single input source, instantiates a data row, and passes the data to a report section.

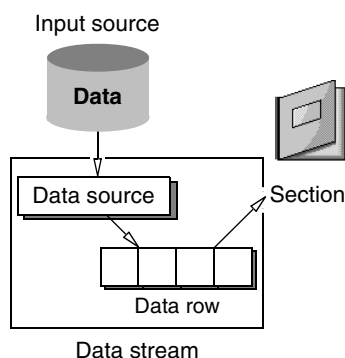


Figure 3-1 Data stream configuration with a single input source

In Figure 3-2, the data stream retrieves data from a single input source, instantiates a data row, filters the data, instantiates a data row for the filtered data, and passes the filtered data to the report section.

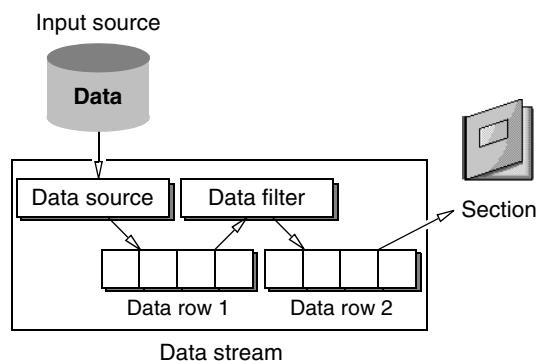


Figure 3-2 Data stream configuration with a single input source and a filter

The data stream in Figure 3-3 uses a single input source and multiple data filters. Each time data is filtered, the data stream instantiates a new data row.

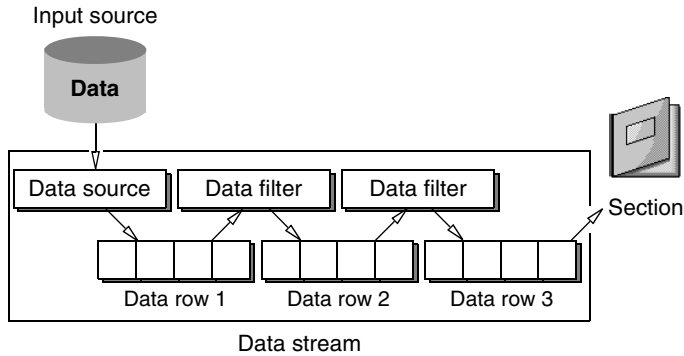


Figure 3-3 Data stream configuration with a single input source and multiple data filters

Figure 3-4 shows a configuration that uses multiple input sources, for example from different databases, and a multiple input data filter.

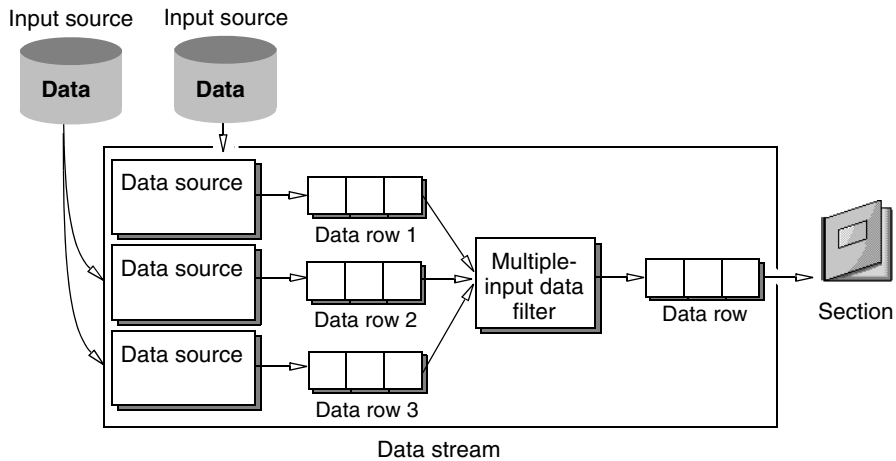


Figure 3-4 Data stream configuration with multiple input sources and a multiple input data filter

Instantiating data adapters

The section instantiates the data adapter that provides data rows to it. If a data stream consists of multiple data adapters, each data adapter instantiates the component that delivers data rows to it. The order in which you set up component references in Report Structure determines the order in which data adapters instantiate.

For example, consider the report design shown in Figure 3-5.

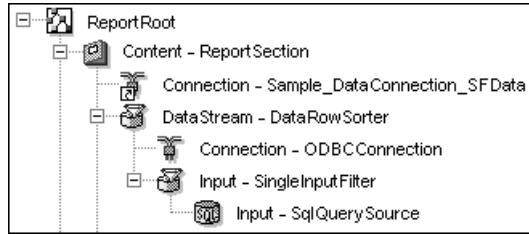


Figure 3-5 Report design

In this report design:

- The report section instantiates MemoryDataSorter.
- MemoryDataSorter instantiates SingleInputFilter.
- SingleInputFilter instantiates the data source, SqlQuerySource.

Figure 3-6 shows the order of instantiation of these components and the direction of data flow.

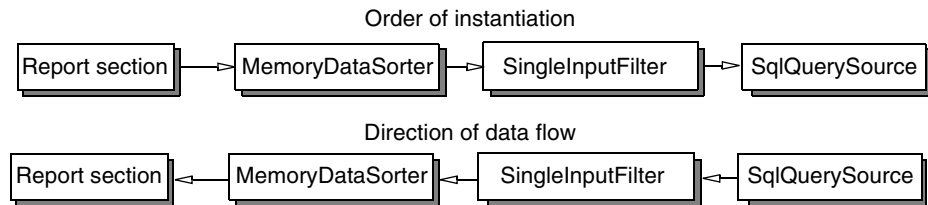


Figure 3-6 Order of instantiation and the direction of data flow

Customizing a data stream by modifying data adapter methods

Typically, you do not need to customize a data stream as e.Report Designer Professional provides rich support for a variety of data source types. If you do want to customize a data stream, you can modify the methods used by the data source components or data filter components. Data source components and data filter components follow a common protocol defined by the abstract base class `AcDataAdapter`.

Adding code for additional data stream tasks

You can add code to perform additional tasks when starting data access, fetching each row, and finishing data access. Table 3-1 describes the methods used for starting to access a data source, finishing data access, and fetching rows. For an example of customizing these methods, see Chapter 6, “Accessing data in a comma-separated values (CSV) text file.”

Table 3-1 Methods for accessing a data source

Method	Description
Start()	Opens the data adapter. If the input source is a spreadsheet, text file, Start() opens the file. If the input source is any other type of input source, Start() opens a cursor. If the input source is a stored procedure, Start() also formats the procedure call for the data source. If the input source is a R/3 data, Start() also processes the RFM export parameter. Override this method to perform a task such as opening a different data source from which the data stream gets data.
Fetch()	Retrieves a single row. To retrieve all rows, the data adapter calls Fetch() repeatedly until there are no more rows. Fetch() returns Nothing after it retrieves all rows. Override this method to populate variables in a data row with the retrieved data row or to provide custom filtering.
Finish()	Closes the data adapter. You can override this method to perform a task such as closing a file you opened in Start().

To change a data row after the data adapter populates it with data from the input record, override the data row's OnRead() method. e.Report Designer Professional calls OnRead() after the data row gets its data. For more information about data rows, see "About data rows," later in this chapter.

Accessing data in non-sequential order

By default, e.Report Designer Professional fetches rows in sequential order. Typically, an input source supports only sequential access to data. A SQL database, for example, returns a set of records based on your query and sends one input record at a time to the data source sequentially.

Sometimes, your report requires random access to data. For example, a report can use the same data multiple times for both tabular and chart formats. A data filter can also require multiple passes over a set of rows for sorting or calculating custom aggregations.

To support random access, AcDataAdapter defines the methods shown in Table 3-2.

Table 3-2 Methods that support random access

Method	Description
CanSeek()	True if the data adapter supports random access
GetPosition()	Returns the position of the next row to fetch

Table 3-2 Methods that support random access

Method	Description
Rewind()	Moves the fetch position to the beginning of the input set
SeekBy()	Moves the fetch position by a given amount relative to the current position
SeekTo()	Moves the fetch position to a given location
SeekToEnd()	Moves the fetch position to one unit past the end of the input set

e.Report Designer Professional provides a data filter class, `AcDataRowBuffer`, that implements the random access methods. You can use this class in a data stream to create a data filter that processes rows in any order.

Using data filters to process rows

A data filter component can compute new values, perform custom lookups, select rows, sort rows, and join data from several data sources. Unlike data source components, data filters are optional. Typically, you use a data filter to process data from a non-SQL database or data that originates from several different external sources.

Use a data filter for a single data source only to process the data rows in ways that the data source does not provide. For example, do not use a sort filter to perform a sort that your data source can do. If your data source does not support returning rows in the row order your report requires you can add a filter to specify that the Actuate software performs the sorting.

A data filter receives data rows from one or more data adapters, processes the data, then passes it to the next data adapter or to the report section. There are two abstract classes of data filters:

- `AcSingleInputFilter`, which accepts input from one data adapter
- `AcMultipleInputFilter`, which accepts input from multiple data adapters

You implement the processing algorithm on data rows in the `Fetch()` method.

Table 3-3 describes the methods for working with a data filter.

Table 3-3 Methods for working with a data filter

Method	Description
Start()	Instantiates and opens the input adapter, the data adapter that provides data rows.

(continues)

Table 3-3 Methods for working with a data filter (continued)

Method	Description
Fetch()	Retrieves a single data row and processes it. Optionally, the data filter creates a new data row.
Finish()	Closes the input adapter.

For examples of creating a data filter, see “Filtering data” and “Working with data from multiple sources,” later in this chapter.

About data rows

A data row consists of variables, each corresponding to a single field or column of the input record. The list of variables defines the data that you can include in the report. A data source component creates a data row instance for each input record it retrieves. A data filter component creates a data row instance for each input record it passes through, possibly modifying the input record first.

If you use Query Editor or Textual Query Editor to write a SQL query, the data source creates the data row. Other data sources, may automatically create the data row. If you use a custom data source or a custom data filter, you must define a custom data row that works with the data source or filter.

A data row component is a class you develop when designing a report. A data row object is an instance of that class that holds the data coming from the data source when you run the report. When you run the report, the data source component instantiates the set of data row objects. In sum, each report design has one data row component, but as you run the report, it can create and process thousands of data row objects, or none, if no rows match the query.

Actuate software processes data rows in the following sequence:

- The data adapter instantiates a data row. If the data adapter is a data source, it instantiates a data row each time it retrieves an input record using `Fetch()`. If the data adapter is a data filter, it instantiates a data row after processing one or more data rows created by one or more data sources or another data filter.
- The data adapter copies the input record’s column values to the data row’s variables, as shown in Figure 3-7. If a report uses a SQL query data source, the `SetupDataRow()` method of the SQL query source performs this task.

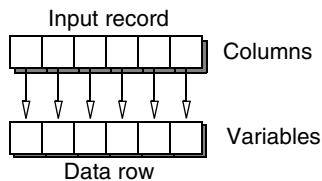


Figure 3-7 Copying an input record’s column values to a data row’s variables

- A data control gets its value from the data row and typically displays it. The control's SetValue() method retrieves from the data row the value of every column, variable name, or method name the control's ValueExp property specifies. The build process generates the code in SetValue().
- The code in SetValue() uses the data row's variables or methods to set the value of the control's DataValue variable, as shown in Figure 3-8. In this example, the generated code in CustomerName::SetValue() assigns the value of customers.customName to the variable in the data row.

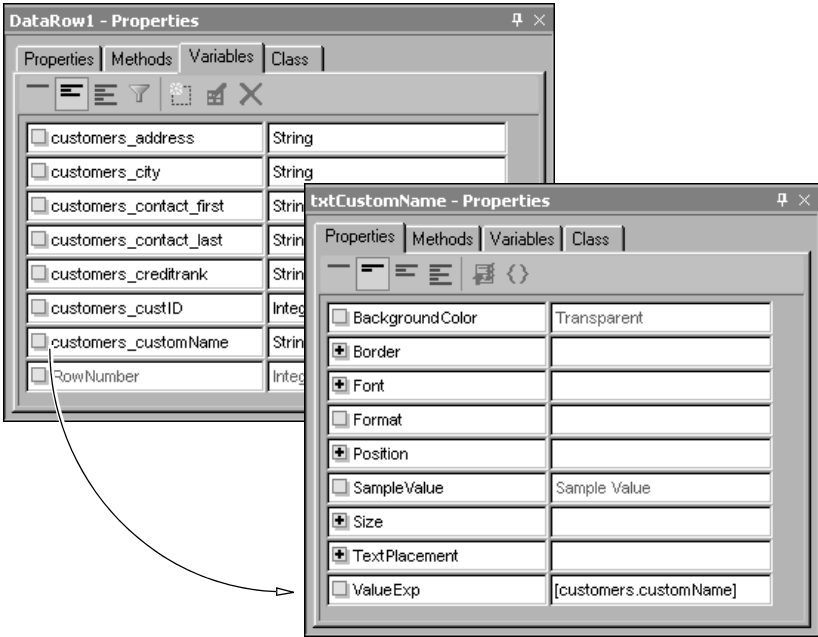


Figure 3-8 Setting the value of the control's Data Value variable

For an example of how to create a custom data row and modifying a data row component, see “Accessing data from a CSV text file” in Chapter 6, “Accessing data in a comma-separated values (CSV) text file.”

Table 3-4 describes how a control implements the core content-creation protocol.

Table 3-4 Methods in the core content-creation protocol for a control

Method	Task
Start()	Typically, a control needs only the default logic. The content of Start() depends on the type of control. Start() handles housekeeping tasks, such as recording sizing information. For some types of controls Start() performs additional types of preparation to display the control.

(continues)

Table 3-4 Methods in the core content-creation protocol for a control (continued)

Method	Task
Start() (<i>continued</i>)	For example, Start() for a graph control determines if the graph type involves a time series and handles special preparation of the x-axis labels if the graph is a pie chart.
BuildFromRow()	Sets the control's value, calling SetValue() as many times as necessary. ■ If a control, such as a line or label control, does not need the data row, BuildFromRow() returns FinishedBuilding. ■ If a control needs only one row, BuildFromRow() sets the value of the control with SetValue(), calls OnRow(), and returns FinishedBuilding. ■ If a control is an aggregate control, it uses multiple data rows, calling SetValue() and OnRow() for each row. BuildFromRow() returns ContinueBuilding.
SetValue()	Sets the control's value. e.Report Designer Professional generates this code during the build phase of creating the report executable (.rox) file. ■ If the control needs no data rows, SetValue() does nothing. ■ If the control uses a single data row, SetValue() sets the value of the control. ■ If the control uses multiple data rows, BuildFromRow calls SetValue() to construct the aggregate control's value().
OnRow()	Override this method to specify additional processing on the control using the data row.
Finish()	For an aggregate control, performs final calculations. For other controls, Finish() does nothing.

Creating a computed field

Typically a field's value is the value of a field in the data source. A computed field is a field whose value is computed from an expression, often involving one or more fields in the data source. For example, if you have fields in the data source providing the type of item, the number of those items ordered and the cost per item, you could have a computed field that calculates the total cost for that type of item.

You can create a computed field using the following techniques:

- Create the computed field as a control. Use this technique if you can use a simple expression to compute the value and only need the computed value in a single control. For more information about creating the computed field as a control, see *Developing Reports using e.Report Designer Professional*.

- Create the computed field as part of the data source query. The query editors for some types of data sources support creating a computed field. Use this technique if a query editor for your data source supports creating computed fields, especially if you need the computed field in multiple controls or report designs. For more information about creating a computed field in a query editor, see the chapter about accessing data from your data source.
- Create a computed data row variable and use it like a data field. Use this technique if you don't have a query editor for your data source that supports creating a computed field and either you need more than a simple expression to compute the value or you need the computed value in a multiple controls or report designs. This section describes how to create a computed data row variable.

A computed data row variable contains a value that is the result of an expression, instead of data stored in the data source. Typically, the expression uses one or more fields in the data source. For example, you can create a data row variable for the number of days an order is late by subtracting the DueDate field's value from the current date. The computed data row variable appears in the Field List with the data source fields. By creating this variable as a computed data row variable, a report developer can treat the variable as if it is a data source field.

To create a computed data row variable, perform the following steps in this order:

- Create the data source component and its data row component.
- Add a variable to the data row to store each computed value. To add a variable, choose New on the Variables page for the data row component.
- Override the OnRead() method of the data row to compute the new values. The data adapter calls OnRead() after it has created a data row. OnRead() sets the data row values.

The following code overrides the data row's OnRead() method to calculate values for the ExtendedCost and DaysLate variables:

```
Sub OnRead()
    Super::OnRead()
    'Set the values of predefined data row values
    Dim ExtendedCost As Double
    ExtendedCost = Cost * Quantity
    DaysLate = Date() - DueDate
    If DaysLate < 0 Then
        DaysLate = 0
    End If
End Sub
```

The data source calls OnRead() once for each data row immediately after the row receives data from the input record. The call to the OnRead() method of the superclass is not necessary but it is a good programming technique.

Accessing other types of data sources

You can specify a connection component for many types of data sources such as various databases. You can also create a custom data source.

How to create a custom data source

- 1 Create a blank report.
 - 1 Choose File→New→Blank Report. Then, Choose OK. The blank report appears in the design perspective.
 - 2 Choose File→Save As.
 - 3 In Save As, name the file and place it in a directory.
- 2 Create a custom data source.
 - 1 In Report Structure, right-click the data stream component and choose Properties.
 - 2 In The Properties page for the component, choose Class.
 - 3 In Super Class, type:
`AcDataSource`
- 3 Create any specialized variables you need for your data source. For example, you can create variables to identify the data source. You can also provide these variables as parameters if you want users to be able to specify their values.
- 4 Define the data row variables. Create a variable for each column or field retrieved from the data source.
- 5 Create controls to display the data in the report. You can choose your data row variables from Fields.
- 6 Override the data stream's Start(), Fetch() and Finish() methods. Use Actuate Basic to specify the actions needed to start using your data source, fetch individual rows, and then finish using your data source.
- 7 Specify whether your custom data source can handle dynamic sorting. If your custom data source can handle queries requesting dynamic sorting, override `AcDataAdapter::CanSortDynamically()` to return True.

You can now run, view, and publish the report. If you will use the data source for other reports, consider publishing your custom data stream to a library.

For an example of creating a custom data source, see Chapter 6, "Accessing data in a comma-separated values (CSV) text file." For more information about using libraries, see *Developing Reports using e.Report Designer Professional*. For more information about `CanSortDynamically()`, see "Sorting data," later in this chapter.

Sorting data

Most reports require data to be sequenced in a particular order. For example, in a sales report you can sort orders alphabetically by customer name. The customers, in turn, you can sort by city. The set of column or field names that determines sort order is called the sort key.

By default, e.Report Designer Professional builds the desired row sorting from the group sections in the report. You can supplement or override this default sorting for your report in the following ways, depending on the sorting capabilities of your data source:

- Rely on your data source to sort the rows before delivering them to e.Report Designer Professional.
Use this technique if your query to the data source returns the data rows in the proper order. The technique for specifying the sort order for your query depends on the type of data source. For example, with a database query, you can add an ORDER BY clause to specify sorting for your query. If the data source produces the data rows in the right sequence for your report, set the report section's sort property to Presorted.

- Request sorting by e.Report Designer Professional after the rows are retrieved from the data source.

You can use the sort filter, *AcDataSorter*, to sort data from custom data sources that cannot perform sorting. Do not use this technique for databases and other data sources that can perform dynamic sorting. Typically data sources are very efficient at sorting.

If you do need internal sorting, specify the desired sorting criteria within the sort filter. The sort filter sorts data using the virtual memory management services of the Actuate iServer System. Sort performance depends on the memory available to store the file in memory. Internal sorting for files that contain more than 10,000 records can adversely affect report generation performance.

- Use the data source's sorting capability, if available.
When developing a report, if you don't know if the data source can perform dynamic sorting, then let the report section's sort property use the default value *AutoSort*. If the sort property is set to *AutoSort*, the Factory calls the method *CanSortDynamically* to determine whether the data source can sort itself. If it can sort itself, database utilities sort the data source. Otherwise, e.Report Designer Professional instantiates a *Memory Data Sorter* filter to sort the data internally. For information about *CanSortDynamically*, see *Programming with Actuate Foundation Classes*.

Sorting data within the data source

For custom data sources, to specify a sort order, you change the data stream component's methods. For other data sources, you can specify how you want your data source to sort values in several ways, listed in order of precedence:

- **Group section's sort key**
The dominant sort expression is provided by a group section's sort key, if any. Each group section contains a sort key that e.Report Designer Professional uses to create the ORDER BY clause for the section's query. For information about overriding the automatic sorting provided by group sections, see "Avoiding sorting by the group section sort key," later in this chapter.
- **OrderBy property of the section component**
You can provide additional sorting instructions through the OrderBy property of the section component. If you also have group sections with sort keys, the sort keys are prepended in order before the code you put as the value of the OrderBy property. Thus the value you use for the OrderBy clause specifies the sorting of rows within the innermost group of your report.

Using the OrderBy property does not require modification of the query. Thus it can be used even for read-only queries from a library and queries specified in methods. You can also use it for data streams that do not contain a query, such as a stored procedure or a flat file.

If you specify a sort order using the report wizard or Sorting and Grouping—Sorting, e.Report Designer Professional handles the sorting by modifying the OrderBy property. You also can specify the OrderBy property directly, as described in "Specifying the sort order using the OrderBy property," later in this chapter.
- **Sorting criteria in the query**
If you use a query to access your data source, you can change the query to specify that the data source sort the data before sending it to e.Report Designer Professional. The procedures for specifying sorting in a query depend on the type of data source. For more information about specifying sorting in the query, see the chapter about that type of data source.

Avoiding sorting by the group section sort key

Group sections contain a sort key. e.Report Designer Professional adds this sort key to the query's ORDER BY clause. Typically you do not override this behavior. If you do not want the sort key added to your ORDER BY clause, you can specify that the data is presorted. Use this functionality when:

- The query from a read-only library requires no modification.
- The query performs specialized sorting, which e.Report Designer Professional should not modify.

You can also modify the default behavior of how group sections sort keys by modifying the query sent to the data source. For information about changing the query sent to the data source, see the appropriate chapter for that data source:

- Chapter 8, “Creating a database or ODBC query.”
- Chapter 5, “Accessing an Actuate Information Object.”

How to use presorted data



- 1 Right-click the report section component in Report Structure and choose Properties.
- 2 In the Properties page for the component, choose Advanced properties. The Properties page displays the advanced properties.
- 3 For the Sorting property, select Presorted from the drop-down list, as shown in Figure 3-9.



Figure 3-9 Selecting the type of sorting

Specifying the sort order using the OrderBy property

You can sort data with the OrderBy property. Sorting data with the OrderBy property does not change the ORDER BY clause of the SQL SELECT statement, if any. e.Report Designer Professional uses the value of the OrderBy property to the specify the sorting of rows within the innermost group produced by the ORDER BY clause.

Use the OrderBy property when:

- The query is from a library, so e.Report Designer Professional cannot modify it.
- The query is defined as text in a method, so e.Report Designer Professional cannot modify it.
- The report design uses a data stream that is not a query, for example a flat file or a stored procedure.

If you specify a sort order using the report wizard or Sorting and Grouping—Sorting, e.Report Designer Professional modifies the OrderBy property accordingly.

How to sort data using the OrderBy property



- 1 Right-click the report section component in Report Structure and choose Properties.
- 2 In the Properties page for the component, choose All properties. The Properties page displays all properties for the component.
- 3 In OrderBy, type a column name or a comma-separated list of column names. Alternatively, select a column from the drop-down list, as shown in Figure 3-10.

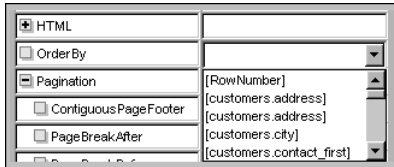


Figure 3-10 Selecting a column to sort

Specifying the sort order for sorting internally

This section describes how to specify sorting when you must sort the rows within e.Report Designer Professional. You create a data filter that sorts data rows by overriding a method. Use the technique described in this section only when sorting within the data source is not feasible.

How to create a sort filter



- 1 If your report design already has a data source, move it to Scratch Pad.
- 2 Place a data adapter in DataStream.
- 3 Select Disk-based Data Row Sorter from the list of data adapters that appears.
- 4 Move the data source from Scratch Pad to the Input slot of the filter, as shown in Figure 3-11. The data source becomes the input source for the filter. The filter is the data adapter that provides a report section with data rows.

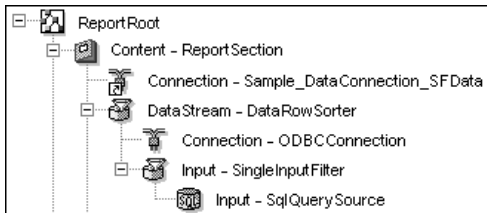


Figure 3-11 A data source component in the input slot of the filter

- 5 Override the filter's Compare() method to specify how to sort the rows. Compare() takes two rows as arguments and returns one of the following values:

- A positive number if the first row goes after the second row
- 0 if both rows are the same
- A negative number if the first row goes before the second row

Call the `CompareKeys()` method to compare field values. `CompareKeys()` performs data type-independent comparisons. For a descending sort order, negate the value `CompareKeys()` returns.

In the following example, `Compare()` compares rows by their state and customer ID:

```
Function Compare( row1 As AcDataRow, row2 As AcDataRow ) As Integer

    Dim cust1 As AcCustomerRow
    Dim cust2 As AcCustomerRow

    Set cust1 = row1
    Set cust2 = row2

    ' Compare the state. CompareKeys() is a predefined method
    Compare = CompareKeys( cust1.State, cust2.State )
    If Compare <> 0 Then
        Exit Function
    End If

    ' Compare the customer ids. Compare two integers by
    subtracting them
    Compare = cust1.ID - cust2.ID
End Function
```

Filtering data

If possible, limit the data returned from your data source by providing filter conditions to the data source. You can use a query, stored procedure code, and other techniques, depending on the type of data source. If limiting data returned from the data source is not feasible, there are two techniques to filter out a row from a set of input records received by e.Report Designer Professional.

- Exclude rows by customizing your data source component.
- Exclude rows by creating a data filter component.

If you encapsulate the filter code in a data filter, you can use the data filter in other data streams. If you publish the filter to a library, you can use the same filter in other report designs.

How to exclude rows by customizing your data source

- 1 Create the data source component.

- 2 If desired, create one or more parameters for users to specify values that the columns in the rows must match.
- 3 Override the data source's Fetch() method to specify which rows to retrieve. In the following example, the Fetch() method of the data source calls Fetch() for the base class to get each row. Fetch() returns the row if the State column contains the correct value. This process repeats until Fetch() retrieves all rows:

```
Function Fetch() As AcDataRow
    Dim row As CustomerRow
    Do
        Set row = Super::Fetch()
        If row Is Nothing Then Exit Function
        ' Test state against value entered by user
        Loop until row.State = StateParam
        Set Fetch = row
    End Function
```

How to exclude rows by creating a select data filter

- 1 If your report design already has a data source, move the data source to Scratch Pad.
- 2 Drag a data filter component from Toolbox—Data and drop it in DataStream, as shown in Figure 3-12.

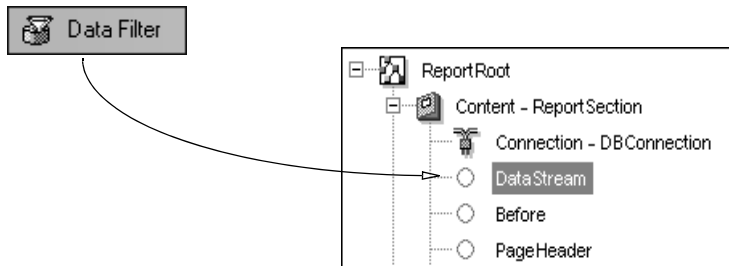


Figure 3-12 Adding a data filter

- 3 In Select Component, select Single Input Filter from the list.
- 4 Move the data source from Scratch Pad to the Input slot of the filter. The data source becomes the input source for the filter, as shown in Figure 3-13.

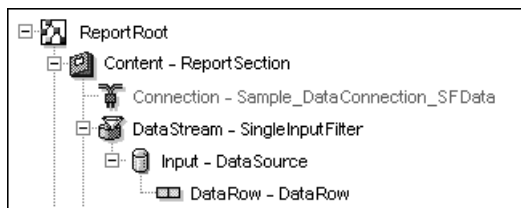


Figure 3-13 A data source in the input slot of the filter

The filter is the data adapter that provides the report section with data rows.

- 5 Override the filter's `Fetch()` method to specify which rows to retrieve. In the following example, the data filter's `Fetch()` method calls the input source's `Fetch()` method to retrieve the row. The `InputAdapter` variable stores a reference to the input source. `Fetch()` returns the row if the `State` column contains the requested value. This process repeats until `Fetch()` retrieves all rows:

```
Function Fetch() As AcDataRow
    Dim row As DataRow
    Do
        Set row = InputAdapter.Fetch()
        If row Is Nothing Then
            Exit Function
        End if
    Loop until row.State = "CA"
    Set Fetch = row
End Function
```

Working with data from multiple sources

If you need to combine data from multiple data sources to create data rows for a single report section, use a multiple input filter or an Actuate information object.

If you purchase the Actuate Data Integration Option, you can create an information object that combines the input from each source. Then, in e.Report Designer Professional you can access the information object as a single data source. For information about accessing information objects, see Chapter 5, "Accessing an Actuate Information Object."

You also can combine input from multiple rows using filters. To retrieve data from multiple sources, use `AcMultipleInputFilter` and override its methods to retrieve and process the data. `AcMultipleInputFilter` accepts input from any number of data adapters, processes the data, and passes it to the next data adapter or to the report. To pass any rows from the source data adapters to the report section, you must override the `Fetch()` method of the multiple input filter.

There are two ways to customize `AcMultipleInputFilter` to process data from two or more input adapters:

- Creating a merge filter to combine the rows of the data sources
A merge filter combines the data rows from two or more input adapters to produce output data rows that can include columns from each input adapter.
- Creating a union filter to process the data sources sequentially
A union filter processes all the data rows from one input adapter before processing data rows from the next input adapter.

You can use the multiple input filter as the input data source of a sort filter. This technique supports sorting the combined set of data rows.

How to create a multiple input filter

- 1 Remove the data stream component from the report, if one exists.
- 2 Drag a data filter component from Toolbox—Data and drop it in DataStream, as shown in Figure 3-14.

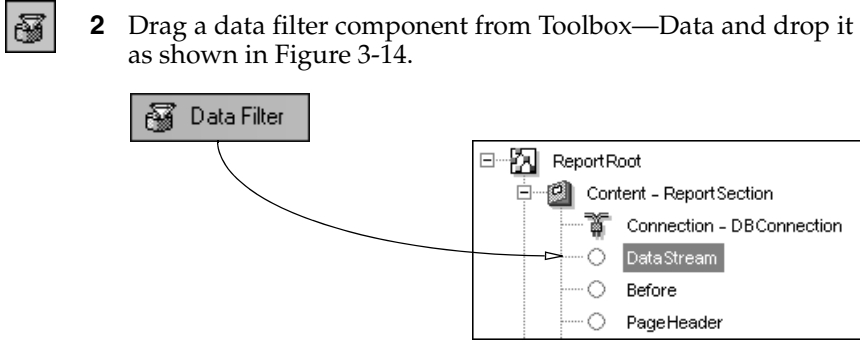


Figure 3-14 Adding a data filter

- 3 In Select Component, select Multiple-Input Filter. Then, choose OK.
- 4 Place a data source into the Input slot of the multiple-input filter.
- 5 Place another data source into the multiple-input filter. The new data source appears as a second Input slot for MultipleInputFilter. Add more data sources as needed. The multiple-input filter gets data from all the data sources.
- 6 Override the multiple-input filter's methods to code a union or merge filter algorithm.

Creating a merge filter to combine the rows of the data sources

Use a merge filter to extract and handle rows from each data source before moving to the next row. You can combine the columns from the input data rows to make a merged output row. You can merge data from multiple data sources or from separate queries to a single data source. This technique emulates a query that the data source cannot support. For example, you can use this technique to emulate joining two tables even if the data source does not have a required relationship to join the tables.

For example, consider how to design a report in which data from two tables must appear in each row. Rows in the report design appear in the following two columns:

- Customers provides customer IDs and names.
- Orders provides order IDs, ship dates, item codes, item price, quantity, and extended price.

To get data from these two tables, the report uses two SQL query data sources. A multiple-input filter creates a new data row by merging the data rows retrieved by each data source. Figure 3-15 shows a report design that uses a merge filter.

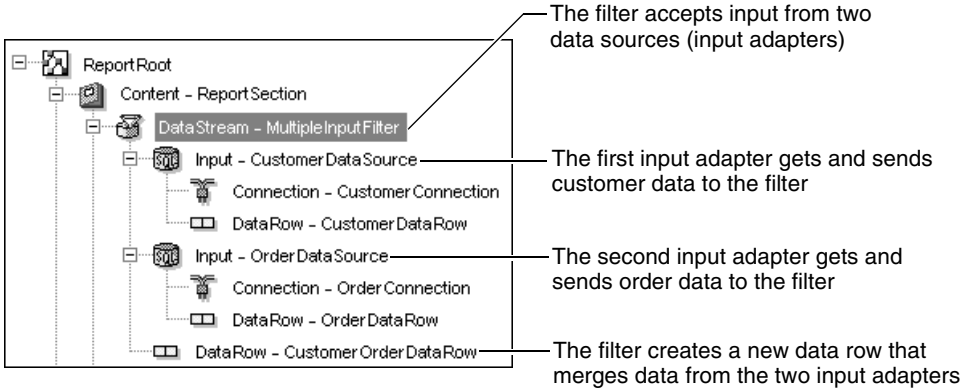


Figure 3-15 Report design with a multiple-input filter

Figure 3-16 shows a portion of report generated from the design described in the preceding example. Each row contains:

- Customer number and name from CustomerDataSource
- Order number, Ship Date, Item Code, Price, Quantity, and Extended Price from OrderDataSource.

CustomerDataSource		OrderDataSource					
Customer	Order	Ship Date	Item Code	Price	Quantity	Extended Price	
101 Signal Engineering	1355	5/1/2004	MPL1632	\$303	1,421	\$430,563	
101 Signal Engineering	1355	5/1/2004	MR3280	\$42	1,442	\$60,564	
101 Signal Engineering	1355	5/1/2004	MRL0840	\$30	1,449	\$43,470	
101 Signal Engineering	1355	5/1/2004	MS0810	\$27	1,411	\$38,097	
101 Signal Engineering	1355	5/1/2004	MSL0840	\$75	1,424	\$106,800	
101 Signal Engineering	1355	5/1/2004	MSL3280	\$210	1,441	\$302,610	
101 Signal Engineering	1355	5/1/2004	MSL3290	\$300	1,434	\$430,200	
Customer	Order	Ship Date	Item Code	Price	Quantity	Extended Price	
102 Technical Specialists Co.	1170	8/26/2004	MP1608x	\$48	126	\$6,048	
102 Technical Specialists Co.	1170	8/26/2004	MPL1632	\$303	124	\$37,572	
102 Technical Specialists Co.	1170	8/26/2004	MR0840	\$15	126	\$1,890	
102 Technical Specialists Co.	1170	8/26/2004	MR1640	\$29	124	\$3,596	
102 Technical Specialists Co.	1170	8/26/2004	MR3290	\$60	126	\$7,560	
102 Technical Specialists Co.	1170	8/26/2004	MRL1610	\$48	127	\$6,096	

Figure 3-16 Generated report with columns from each data source

How to merge rows from several data sources

- 1 Create a multiple-input filter component and several data sources, as specified in steps 1 to 5 of “How to create a multiple input filter,” earlier in this chapter.
- 2 Drag a data row component to the data row slot of the multiple-input filter. Creating the data row also creates a read-only variable for the row number.
- 3 Define public instance variables for each field from the data sources.
 - 1 Right-click the data row component for the filter component and choose Properties.
 - 2 In Properties, choose Variables.
 - 3 In Properties—Variables, choose New Variable.
 - 4 In Class Variable, type the name of the variable. Typically, this name is the same as the data field name.
 - 5 Select the data type of the variable from the drop-down list. Use the Actuate Basic data type that maps to the data source data type. Then, choose OK.
- 6 Repeat steps 3 to 5 for the other data source fields.

Figure 3-17 shows the variables for this example.



☐ CustId	Integer
☐ CustomName	String
☐ ExtPrice	Integer
☐ ForecastShipDate	Date
☐ ItemCode	String
☐ OrderId	Integer
☐ Price	String
☐ Quantity	Integer
☐ RowNumber	Integer

Figure 3-17 Variables for each field

- 4 Override the filter’s Start() method to create an input adapter for each data source and get a row from the first data source:

```
Function Start( ) As Boolean
    Start = Super::Start( )
    Set CustomerQuery = InputAdapters.GetAt( 1 )
    Set OrderQuery = InputAdapters.GetAt( 2 )
    Set CustomerInRow = CustomerQuery.Fetch( )
End Function
```

- 5 Override the filter’s Fetch() method to retrieve one row of data from the second input adapter that matches the row from the first input adapter, create a new data row with the merged data, and pass the data row to the report. This process continues until there are no more rows to retrieve.

```

Function Fetch( ) As AcDataRow
    Dim OutRow As CustomerOrderDataRow
    Dim OrderInRow As OrderDataRow

    If CustomerInRow Is Nothing Then
        Exit Sub
    End If
    Set OrderInRow = OrderQuery.Fetch( )
    If OrderInRow Is Nothing Then
        Exit Sub
    End If

    ' Loop until we find a matching pair of rows.
    While (CustomerInRow.customers_custId <>
        OrderInRow.orders_custId)
        While (CustomerInRow.customers_custId <
            OrderInRow.orders_custId)
            ' Advance through Customers.
            Set CustomerInRow = CustomerQuery.Fetch( )
            If CustomerInRow Is Nothing Then
                Exit Sub
            End If
            While (OrderInRow.orders_custId <
                CustomerInRow.customers_custId)
                ' Advance through Orders.
                Set OrderInRow = OrderQuery.Fetch( )
                If OrderInRow Is Nothing Then
                    Exit Sub
                End If
            Wend
        Wend

        ' If we get here, we found a matching pair of rows.
        ' Create and populate a new output row.
        Set OutRow = New CustomerOrderDataRow
        OutRow.CustId = CustomerInRow.customers_custId
        OutRow.CustomName = CustomerInRow.customers_customName
        OutRow.OrderId = OrderInRow.orders_orderID
        OutRow.ForecastShipDate = OrderInRow.orders_forecastShipDate
        OutRow.ItemCode = OrderInRow.items_itemcode
        OutRow.Quantity = OrderInRow.items_quantity
        OutRow.Price = OrderInRow.items_pricequote
        OutRow.ExtPrice = OrderInRow.extprice

        ' Process the output row.
        AddRow( OutRow )

        ' Return the output row.
        Set Fetch = OutRow
    End Function

```

Creating a union filter to process the data sources sequentially

A union filter processes all rows in a data source before moving to the next data source in the filter. For example, in the report design shown in Figure 3-18, the multiple-input filter processes all the data from the first data source, then all data from the second data source.

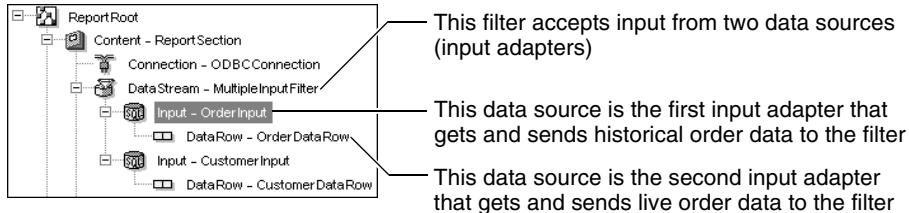


Figure 3-18 Report Structure that includes a union filter

Figure 3-19 shows part of the generated report. The report displays all the data from the first data source, then all data from the second source.

Ship Date	Order	Item Code	Price	Quantity	Extended Price	
5/1/1997	750	MV1632	\$150	2,300	\$345,000	Historical order data
5/1/1998	1055	MS3240	\$77	3,000	\$231,000	
5/1/1999	1455	MS1680	\$82	3,896	\$319,472	
5/1/1999	1455	MRL0480	\$18	3,880	\$69,840	
5/1/1999	1455	MR1690	\$39	3,834	\$149,526	
5/1/1999	1455	MPL2032	\$650	3,836	\$2,493,400	
5/1/2000	5050	MS1610	\$60	4,000	\$240,000	Live order data
5/1/2000	5050	MRL3240	\$61	3,000	\$183,000	
2/4/2004	1455	MS1680	\$82	3,896	\$319,472	
2/4/2004	1455	MRL0480	\$18	3,880	\$69,840	
2/4/2004	1455	MR1690	\$39	3,834	\$149,526	
2/4/2004	1455	MPL2032	\$650	3,836	\$2,493,400	

Figure 3-19 Report output showing the data from one input source followed by the data from the second input source

How to access multiple data sources sequentially

- 1 Create a multiple-input filter component and several data sources, as specified in steps 1 to 5 of “How to create a multiple input filter,” earlier in this chapter.
- 2 Drag a data row component to the data row slot of the multiple-input filter. Creating the data row also creates a read-only variable for the row number.
- 3 Define public instance variables for each field from the data sources.
 - 1 Right-click the data row component for the filter component and choose Properties.



- 2 In Properties, choose Variables.
- 3 In Properties—Variables, choose New Variable.
- 4 In Class Variable, type the name of the variable. Typically, this name is the same as the data field name.
- 5 Select the data type of the variable from the drop-down list. Choose the Actuate Basic data type that maps to the data source data type. Then, choose OK.
- 6 Repeat steps 3 to 5 for the other data source fields.

Figure 3-20 shows the variables for this example.

☐ CustId	Integer
☐ CustomName	String
☐ ExtPrice	Integer
☐ ForecastShipDate	Date
☐ ItemCode	String
☐ OrderId	Integer
☐ Price	String
☐ Quantity	Integer
☐ RowNumber	Integer

Figure 3-20 Variables for each field

- 4 Override the filter's Start() method to create an input adapter for each data source and get a row from the first data source.

```
Function Start( ) As Boolean
    Start = Super::Start( )
    Set HistoricalQuery = InputAdapters.GetAt( 1 )
    Set LiveQuery = InputAdapters.GetAt( 2 )
    Set HistoricalInRow = HistoricalQuery.Fetch( )
End Function
```

- 5 Override the filter's Fetch() method to retrieve each row of data from the first input adapter and pass it to the report. When the first input adapter retrieves all rows, retrieve rows from the second input adapter.

```
Function Fetch( ) As AcDataRow
    Dim OutRow As OrderDataRow
    Dim LiveInRow As LiveDataRow

    ' Assumption:
    ' All live data is newer than all historical data
    If HistoricalInRow Is Nothing Then
        Set LiveInRow = LiveQuery.Fetch( )
        If LiveInRow Is Nothing Then
            Exit Sub
        End If
    End If
```

```

Else
    ' Assign output row with live row values
    Set OutRow = New OrderDataRow
    OutRow.OrderId = LiveInRow.orders_orderID
    OutRow.ForecastShipDate =
        LiveInRow.orders_forecastShipDate
    OutRow.ItemCode = LiveInRow.items_itemcode
    OutRow.Quantity = LiveInRow.items_quantity
    OutRow.Price = LiveInRow.items_pricequote
    OutRow.ExtPrice = LiveInRow.extprice
End If
Else
    ' Assign output row with historical row values
    Set OutRow = New OrderDataRow
    OutRow.OrderId = HistoricalInRow.orders_orderID
    OutRow.ForecastShipDate =
        HistoricalInRow.orders_forecastShipDate
    OutRow.ItemCode = HistoricalInRow.items_itemcode
    OutRow.Quantity = HistoricalInRow.items_quantity
    OutRow.Price = HistoricalInRow.items_pricequote
    OutRow.ExtPrice = HistoricalInRow.items_extprice
    ' Get the next row
    Set HistoricalInRow = HistoricalQuery.Fetch( )
End If

' Process the output row.
AddRow( OutRow )

' Return the output row.
Set Fetch = OutRow
End Function

```


Displaying data rows

This chapter contains the following topics:

- About displaying data rows
- Modifying a report design to display data rows
- Specifying the columns to display data rows
- Modifying font attributes for displaying data rows

About displaying data rows

You can view the data rows in your report section in a simple format, even if you have not designed a report layout. Using this capability, you can see the data that your report receives from the data source without any formatting. The procedure for displaying data rows depends on the type of data source:

- Database or ODBC queries
If you are using a graphical database or ODBC query, see “Previewing the rows that a database or ODBC query returns” in Chapter 8, “Creating a database or ODBC query.”
- Information object
If you are using an information object as your data source, see “Setting up the report design to access information objects” in Chapter 5, “Accessing an Actuate Information Object.”
- All other data sources
This chapter discusses the procedures for displaying data rows from all other data sources that e.Report Designer Professional supports.

Using this technique, you can examine the data that is returned from your data source or constructed programmatically.

e.Report Designer Professional displays the data rows in columns and rows. This format provides a single row of output for each data row. Any sorting and grouping that is defined in the ROD does not affect the data rows in this display format. By default, the labels for the columns show the table name and column name from the data source. e.Report Designer Professional truncates the table name and column name if the total length exceeds the length of the data row variable. Figure 4-1 shows the default data row display format.

customers.addr	customers.ci	customers.contact_fi	customers.contact_l	customers.creditra	customers.cus	customers.custom
18731 Douglas	Allentown	Betty	Murphy	A	182	CompuBoards
18731 Douglas	Allentown	Betty	Murphy	A	182	CompuBoards
18731 Douglas	Allentown	Betty	Murphy	A	182	CompuBoards
18731 Douglas	Allentown	Betty	Murphy	A	182	CompuBoards
18731 Douglas	Allentown	Betty	Murphy	A	182	CompuBoards
18731 Douglas	Allentown	Betty	Murphy	A	182	CompuBoards
18731 Douglas	Allentown	Betty	Murphy	A	182	CompuBoards
18731 Douglas	Allentown	Betty	Murphy	A	182	CompuBoards
18731 Douglas	Allentown	Betty	Murphy	A	182	CompuBoards
18731 Douglas	Allentown	Betty	Murphy	A	182	CompuBoards
18731 Douglas	Allentown	Betty	Murphy	A	182	CompuBoards
18731 Douglas	Allentown	Betty	Murphy	A	182	CompuBoards
16780 Pompton	Brickhaven	Leslie	Taylor	B	211	InfoSpecialists Inc.
16780 Pompton	Brickhaven	Leslie	Taylor	B	211	InfoSpecialists Inc.

Figure 4-1 Default data row display format

You display data rows by creating a report object design (.rod) file in which the ReportType property setting is InformationObject. When you choose Build and Run, e.Report Designer Professional creates a data object executable (.dox) file.

This file is like a report object executable (.rox) file, except that it contains data without formatting.

To display data rows:

- Create or open the report design that contains the data row component to display.
You can use an existing report design or create a new one that accesses your data source or programmatically generates data rows. If you create a new report design, you must specify the data source component, but creating a report layout is not necessary.
- Modify the report design to display data rows.
To display data rows, you modify a report root component to set its ReportType property to InformationObject.
- Determine which columns to include and the order in which they display.
- Modify the font attributes, if necessary.
- Build and run the report object design (.rod) file to generate a data object executable (.dox) file.
If Requester appears, make your selections and choose OK. e.Report Designer Professional displays the data rows. Use the scroll bars, if necessary, to see all data row variables and all data rows.

Modifying a report design to display data rows

To display data rows, set the ReportType property of the report root component to InformationObject. You can create a new report or modify an existing report to use this setting. The report design preserves all report formatting, grouping, and sorting information, but does not use this information to display data rows. To later generate and display formatted report output, set the ReportType property to FormattedReport.

The first section component must be a report section. Displaying data rows uses the data source that is in the first report section's DataStream slot. If the report design contains additional nested report sections, the information about those report sections remains in the report design, but e.Report Designer Professional ignores them. If you later set the ReportType property to FormattedReport, the generated report object instance (.roi) file contains all report sections.

If the first section is a conditional section, group section, parallel section, or sequential section, you must remove that section from the report design before you can display data rows. You can only display data rows from a report section. You can also use this method to view a report section that is nested within another report section. You can use Scratch Pad to save the original component.

How to display data rows when the first section component is a report section



- 1 In Report Structure, select the report component.
- 2 Choose View→Properties.
- 3 In the Properties page, in ReportType, select InformationObject from the drop-down list, as shown in Figure 4-2.



Figure 4-2 Selecting InformationObject

- 4 Choose Report→Build and Run. If Requester appears, make your selections and choose OK. e.Report Designer Professional generates the data object executable file, runs it, and displays the data rows in Report Viewer.

How to display data rows for a section component that is not the first in a design

- 1 Choose View→Scratch Pad. Scratch Pad appears, as shown in Figure 4-3.



Figure 4-3 Scratch Pad

- 2 Replace the first section component with a report section component.
 - 1 Drag the first section component from Report Structure and drop it in Scratch Pad, as shown in Figure 4-4.



Figure 4-4 Moving the first section component

- 2 In Scratch Pad, expand the section component until you see the report section for which you want to display data rows.
 - 3 Drag the report section from Scratch Pad and drop it in the report component, as shown in Figure 4-5.

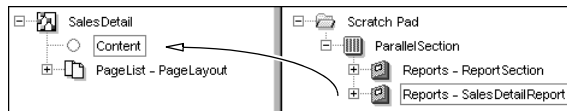


Figure 4-5 Moving the desired report section

Report Structure looks like the one in the Figure 4-6.

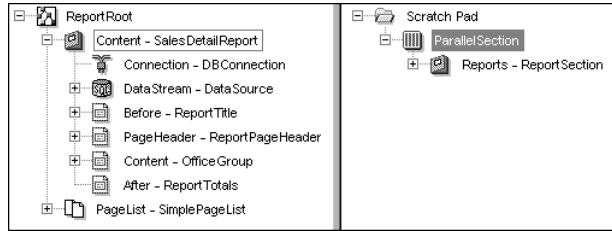


Figure 4-6 Report Structure with the desired report section as the first section component

- 3 Choose View→Properties.
- 4 In the Properties page, in ReportType, select InformationObject from the drop-down list, as shown in Figure 4-7.



Figure 4-7 Selecting InformationObject

- 5 Choose Report→Build and Run. If Requester appears, make your selections and choose OK.
- 6 If desired, replace the report section component with the original section component.
 - 1 Drag the report section component from Report Structure and drop it in Scratch Pad to return it to its original place within the section component, as shown in Figure 4-8.

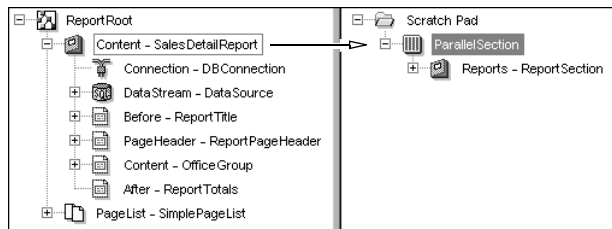


Figure 4-8 Returning the report section component

- 2 Drag the original section from Scratch Pad and drop it in the first Content slot in the report component, as shown in Figure 4-9.

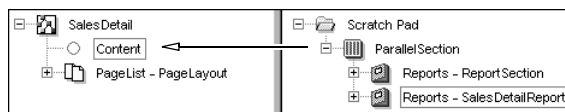


Figure 4-9 Returning the original section

Specifying the columns to display data rows

Each data row consists of a set of variables that contain the data from the data source. You can modify the data source variables in the data rows. You can add or delete computed data row variables. You also can change the order of the data row variables.

Changes that you make to the data row variables affect how the data rows appear when you display data rows. If you add or delete the data row variables, these changes affect the fields that are available to the report and their properties. Modifying the data row variables changes the default attributes of the fields. These changes also show when you display data rows. When you display data rows, e.Report Designer Professional displays each data row variable as a column. Changing the order of the data row variables changes the order of the corresponding columns when you display data rows. The data row variable's attributes control the column label and the column length for that variable.

Changing the order of columns

For any report design, you can use Data Row Editor to display the data row component. If the ReportType property is InformationObject, you can use Data Row Editor to change the order in which data row variables appear as columns when you display the data rows. The default behavior displays data source table names and column names in alphabetical order. In Figure 4-10, Data Row Editor displays the available data row variables for a report design.

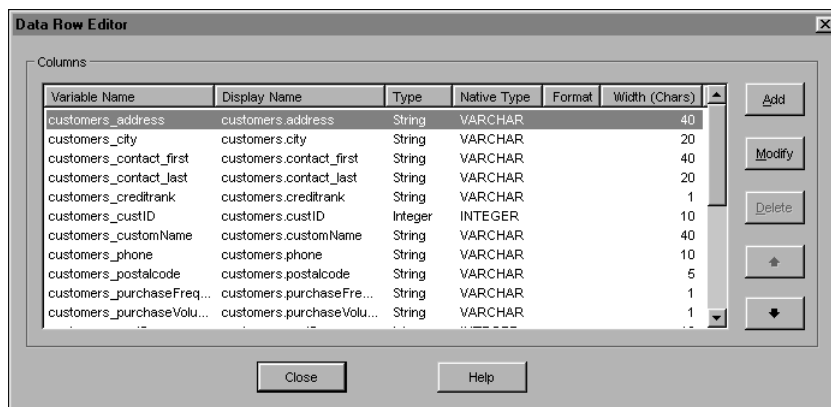


Figure 4-10 Available data row variables

How to change the order in which columns appear when displaying data rows

e.Report Designer Professional displays data row variables as columns in the order in which they appear, from top to bottom, in Data Row Editor.

- 1 Expand the DataStream component, as shown in Figure 4-11.

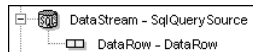


Figure 4-11 The DataStream component

- 2 Right-click the DataRow component and choose Data Row Editor. Data Row Editor displays the available data row variables, as shown in Figure 4-12.

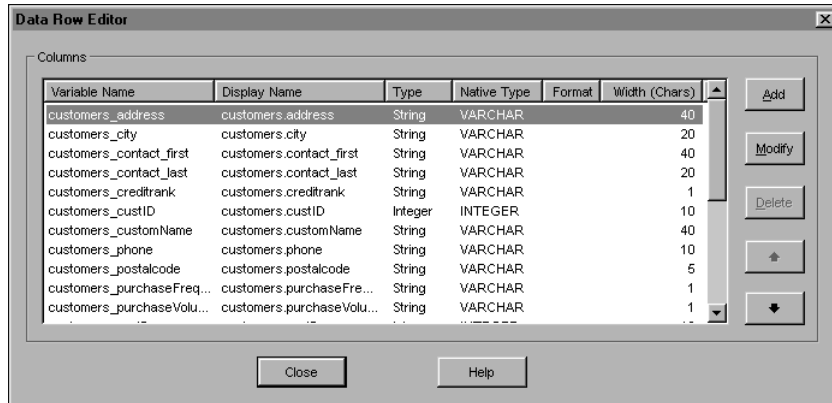


Figure 4-12 Available data rows

- 3 Select a data row variable, then use the up and down arrows to reorder the variable. Figure 4-13 shows the Data Row Editor after you move customers_address to the third position and customers_city to the fourth position.

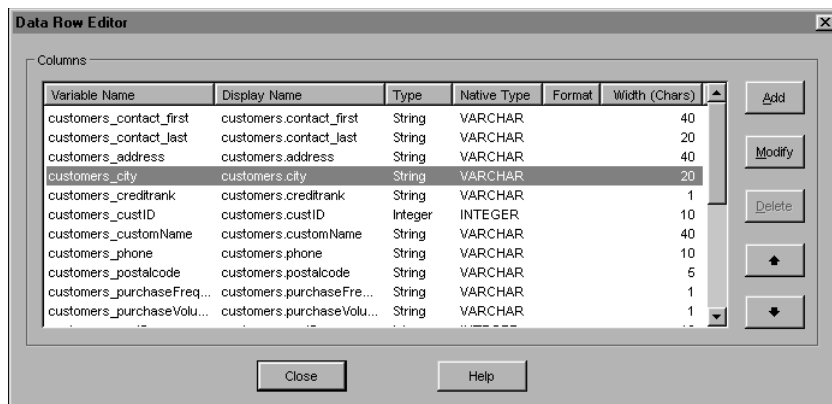


Figure 4-13 Reordered data row variables

Choose Close.

- 4 Choose Report→Build and Run. If Requester appears, make your selections and choose OK. e.Report Designer Professional generates the data object

executable file, runs it, and displays the data rows in Report Viewer. Figure 4-14 shows the columns, reordered as specified in Data Row Editor.

customers.contact_first	customers.contact_la	customers.address	customers.city	customers..
Betty	Murphy	18731 Douglas Av.	Allentown	182
Leslie	Taylor	16780 Pompton St.	Brickhaven	211
Peter	Chandler	4030 Long Airport Avenue	Newark	118
Dan	Chandler	20789 Baden Av.	Brickhaven	212
Dan	Firrelli	3764 Baden Av.	Newark	122
Jose	Barajas	5420 Ingle Ln.	New Rochelle	157
Julie	King	25593 South Bay Ln.	Bridgewater	142
Wai Chung	Tam	613 Furth Circle	New Rochelle	156
Valarie	Nelson	5677 Strong St.	Bridgewater	141
Diane	Chandler	23727 Kingston Rd.	New Bedford	204
Kyung	Tseng	4658 Baden Av.	Cambridge	199
Kwai	Fong	2732 Spinnaker Dr.	Albany	166
Stan	Smith	2389 Park Ln.	Sneadon's Landing	161
Maria	Avila	7497 Baden Av.	White Plains	128
Julie	Brown	7734 Strong St.	Trenton	148

Figure 4-14 Report Viewer, showing the reordered columns

Modifying, adding, and deleting columns

You can use Column Editor to modify a data row variable or add a computed data row variable to the data row component. Data row variables that you add using Column Editor appear in Data Row Editor. e.Report Designer Professional also displays the new data row variables as columns when it displays the data rows.

In Column Editor, you also can modify some attributes of existing variables. The attributes you can modify vary depending on whether the column is a computed field. Figure 4-15 shows Column Editor displaying the attributes for a data row variable.

Column Editor

Variable name:customers_creditrank

Display name:customers_creditrank

Table.col:[customers_creditrank]

Data type:String

Native data type:VARCHAR

Format:

☒ Column can be used as custom filter

Width:1

(number of characters)

☐ Word wrap

Text format

☒ Plain

☐ RTF

☐ HTML

e.Analysis option

☒ Automatic

☐ Dimension

☐ Measure

Expression

OK

Cancel

Help

Figure 4-15 Attributes for a data row variable

Variables that you add to the data row component using Column Editor also appear in Data Row Editor. You can use Data Row Editor to delete a variable you add using Column Editor. Table 4-1 describes the column attributes that can appear in Column Editor.

Table 4-1 Column Attributes

Column attribute	Description	Action
Variable name	Name of data row variable.	Editable for computed data row variables.
Display name	Optional column heading that appears instead of the default heading when displaying data rows.	Editable. Column Editor does not support using the following characters in Display name: \\ / : ? "
Table.col	Name of the column in the data source. This value only appears if e.Report Designer Professional retrieves the value from the data source.	Not editable.
Data type	Actuate Basic data type of the data row variable. For computed data row variables, there is no default data type. For other data row variables, e.Report Designer Professional sets the data type.	Editable if the data row variable is computed or if the data source data type can map to multiple Actuate Basic data types. Changing the data type affects the control type for all controls using the data row variable.
Native data type	Data type that is specified in a data source.	e.Report Designer Professional maps the native data type to an Actuate Basic data type. For example, a Double in a data source can map to Actuate Basic data type Double, Integer, Currency, or String.
Format	Data format.	Editable. For example, to display a date as the day of the week, month, day, and four-digit year, select Long Date from the drop-down list.
Width	Column display width in number of characters. e.Report Designer Professional uses this value and the font size to calculate the column width.	If the columns appear too large or too small, adjust this value. If you set this attribute value too low, the column truncates characters.
Column can be used as custom filter	This attribute is not used in displaying data rows.	None.

(continues)

Table 4-1 Column Attributes (continued)

Column attribute	Description	Action
Expression	Actuate Basic expression for a computed data row variable.	Type an expression that applies to a row. For example, you can type the following Actuate Basic expression: [items_quantity] * 2 This field does not support aggregate expressions, such as: Sum([items_quantity])
e.Analysis option	This attribute is not used in displaying data rows.	None.

How to modify a column

- 1 Expand the DataStream slot, as shown in Figure 4-16.

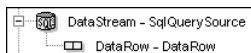


Figure 4-16 The DataStream slot

- 2 Right-click the DataRow component and choose Data Row Editor. Data Row Editor appears, displaying the available columns.
- 3 Select a column, as shown in Figure 4-17.

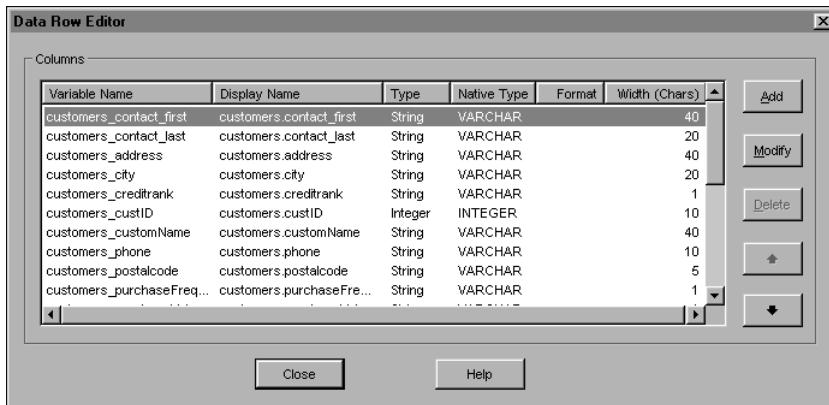


Figure 4-17 Selecting a column

- 4 Choose Modify. Column Editor appears. For the customers_contact_first column that appears in Figure 4-18, the Variable name, Table.col, Data type, Native Data Type, and Expression fields are not editable.

The Column Editor dialog box is shown with the following settings:

- Variable name: customers_contact_first
- Display name: customers_contact_first
- Table.col: [customers_contact_first]
- Data type: String
- Native data type: (empty)
- Format: (empty)
- Column can be used as custom filter: ☒
- Word wrap: ☐
- Width: 40 (number of characters)
- Text format: Plain (selected), RTF, HTML
- e. Analysis option: Automatic (selected), Dimension, Measure
- Expression: (empty)

Buttons: OK, Cancel, Help

Figure 4-18 Column Editor, showing attributes for a column

- 5 Modify the available fields and options as desired. Figure 4-19 shows the display name and width properties that are modified for the customers_contact_first column.

The Column Editor dialog box is shown with the following settings:

- Variable name: customers_contact_first
- Display name: First Name
- Table.col: [customers_contact_first]
- Data type: String
- Native data type: (empty)
- Format: (empty)
- Column can be used as custom filter: ☒
- Word wrap: ☐
- Width: 15 (number of characters)
- Text format: Plain (selected), RTF, HTML
- e. Analysis option: Automatic (selected), Dimension, Measure
- Expression: (empty)

Buttons: OK, Cancel, Help

Figure 4-19 Modifying the display name and width properties

Choose OK. Data Row Editor appears as shown in Figure 4-20, showing the new display name and width for the customers_contact_first variable.

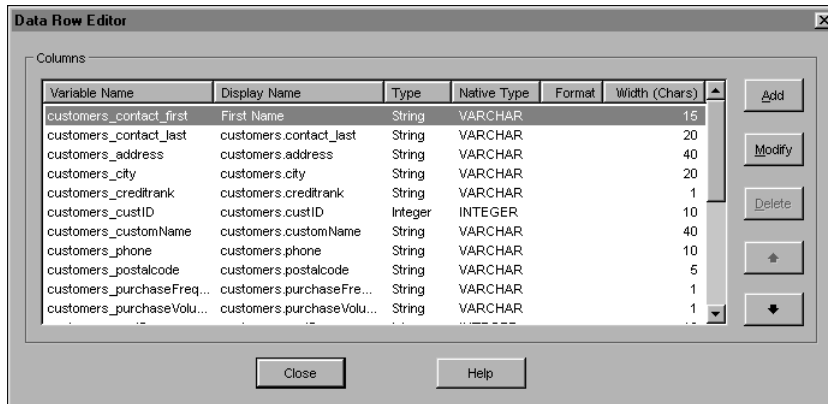


Figure 4-20 A data row variable, showing a new display name and width

- Repeat steps 3 through 5 for each column that you want to modify. Then, choose Close.
- Choose Report→Build and Run. If Requester appears, make your selections and choose OK. e.Report Designer Professional generates the data object executable file, runs it, and displays the data rows in Report Viewer. Figure 4-21 shows the effect of changing the display names and widths of several columns.

First Name	Last Name	Address	City	Rank	ID	Company	Phone
Betty	Murphy	18731 Douglas Av.	Allentown	A	182	CompuBoards	2155554753
Leslie	Taylor	16780 Pompton St.	Brickhave	B	211	InfoSpecialists Inc.	6175558428
Peter	Chandler	4030 Long Airport	Newark	B	118	Computer Systems Corp.	2015558624
Dan	Chandler	20789 Baden Av.	Brickhave	A	212	SigniEngineering Co.	6175559035
Dan	Firrelli	3764 Baden Av.	Newark	A	122	TeleMicroSystems	2015555171
Jose	Barajas	5420 Ingle Ln.	New	A	157	Computer Engineering	9145557064
Julie	King	25593 South Bay Ln.	Bridgewater	B	142	SigniSpecialists	2035552570
Wai Chung	Tam	613 Furth Circle	New	B	156	Advanced Design Corp.	9145556707
Valarie	Nelson	5677 Strong St.	Bridgewater	A	141	Design Design	2035551450
Diane	Chandler	23727 Kingston Rd.	New	B	204	Computer MicroSystems Corp.	5085557307
Kyung	Tseng	4658 Baden Av.	Cambridge	A	199	Technical Specialists Corp.	6175555910
Kwai	Fong	2732 Spinnaker Dr.	Albany	A	166	Advanced Solutions Inc.	5185559644
Stan	Smith	2399 Park Ln.	Sneadon	A	161	Exosoft Corp.	9145557172
Maria	Avila	7497 Baden Av.	White	A	128	Computer MicroSystems Corp.	9145558205
Julie	Brown	7734 Strong St.	Trenton	A	148	Design Systems Inc.	6095551386

Figure 4-21 Report Viewer, showing the new display names and widths

How to add a column to a data row component

- Expand the DataStream component.
- Right-click the DataRow component and choose Data Row Editor. Available columns appear in Data Row Editor, as shown in Figure 4-22.

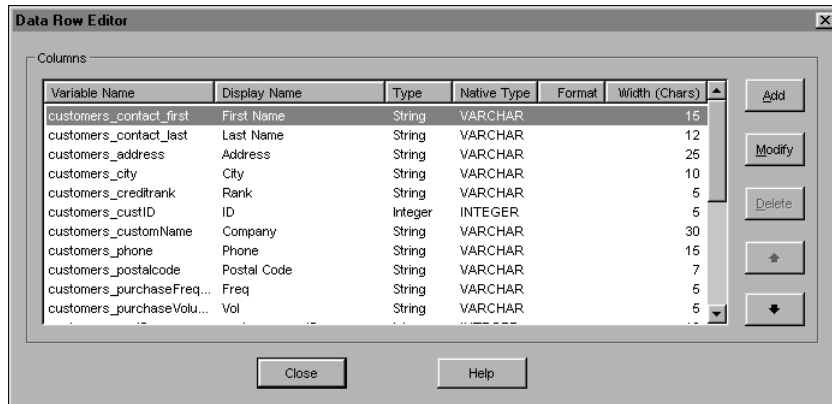


Figure 4-22 The available columns

- 3 Choose Add.
- 4 On Column Editor, provide values for the available fields and select options. Figure 4-23 shows an example of specifying a variable.

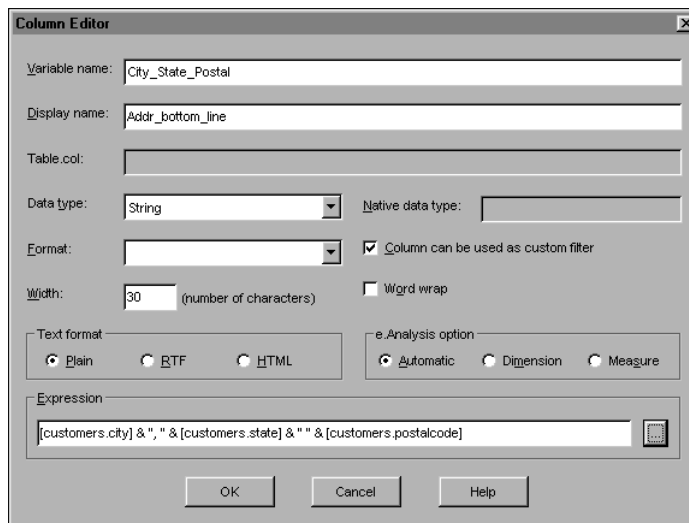


Figure 4-23 Setting attributes for a variable

Choose OK.

The column appears in Data Row Editor. By default, the new variable is added to the bottom of the list, as shown in Figure 4-24.

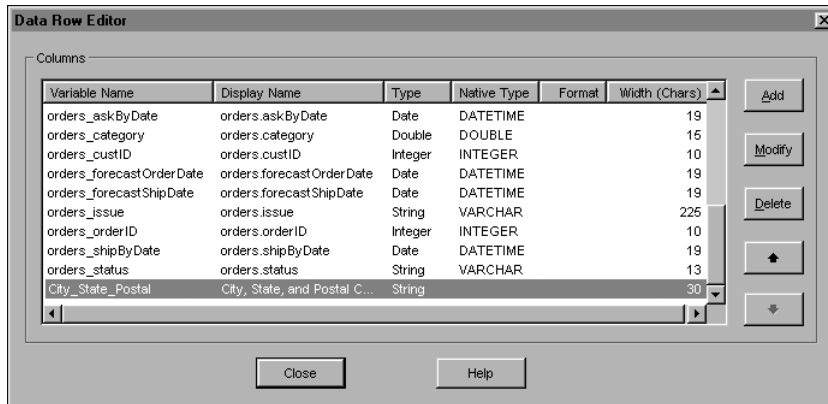


Figure 4-24 Data Row Editor, showing the new variable

- Repeat steps 3 and 4 for any additional columns that you want to add. Then, choose Close.
- Choose Report→Build and Run. If Requester appears, make your selections and choose OK. e.Report Designer Professional generates the data object executable file, runs it, and displays the data rows in Report Viewer. Figure 4-25 shows the additional data row variable. The new data row variable appears to the right of all existing data row variables. You can reorder the data row variables, as described in “Changing the order of columns,” earlier in this chapter.

orders.ord	orders.shipByDate	orders.status	City, State, and Postal Code
1075	2/4/2004	Closed	Allentown, PA 70267
1080	2/13/2004	Closed	Brickhaven, MA 58339
1085	2/21/2004	Closed	Newark, NJ 94019
1095	2/10/2004	Closed	Brickhaven, MA 58339
1100	5/8/2004	Cancelled	Newark, NJ 94019
1105	4/23/2004	In Evaluation	New Rochelle, NY 12066
1110	6/14/2004	In Evaluation	Bridgewater, CT 97562
1115	8/17/2004	Open	New Rochelle, NY 12066
1120	9/3/2004	Open	Bridgewater, CT 97562
1125	3/1/2004	Closed	New Bedford, MA 50553
1130	7/15/2004	Open	Cambridge, MA 51247
1140	6/15/2004	Open	Albany, NY 16381
1142	4/15/2004	Open	Sneadon's Landing, NY 21739
1145	7/18/2004	Open	White Plains, NY 24067
1150	4/1/2004	Closed	Trenton, NJ 94217

Figure 4-25 Report Viewer, showing the new variable

How to delete a column

You can delete columns that you added to the data row.

- Expand the DataStream component.
- Right-click the DataRow component and choose Data Row Editor. Data Row Editor displays the available columns, as shown in Figure 4-26.

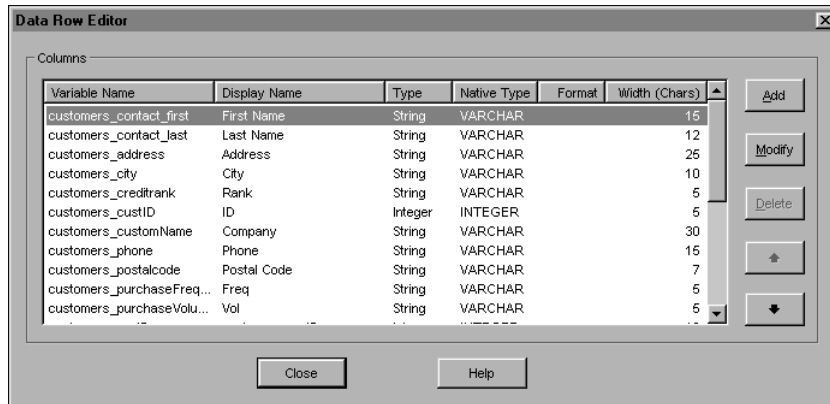


Figure 4-26 The available columns

- 3 To delete a column, select the column. If the column can be deleted, the Delete button is enabled, as shown in Figure 4-27.

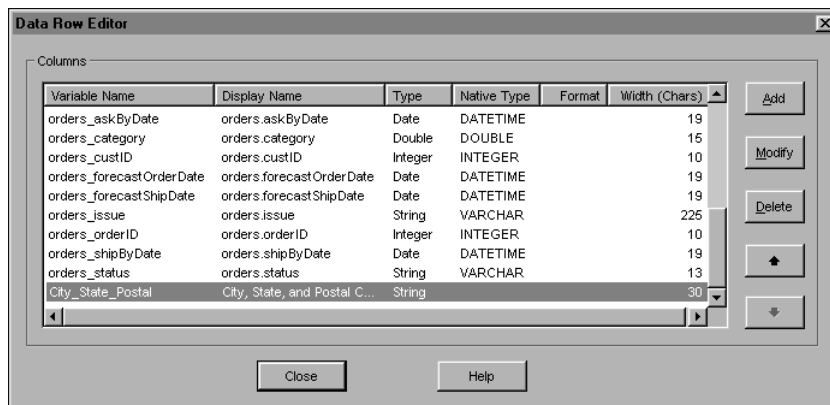


Figure 4-27 A selected column showing the enabled Delete button

Choose Delete. The column is deleted from Data Row Editor. Choose Close.

- 4 Choose Report→Build and Run. If Requester appears, make your selections and choose OK. e.Report Designer Professional generates the data object executable file, runs it, and displays the data rows in Report Viewer. The displayed data row does not include the deleted column.

Modifying font attributes for displaying data rows

In a report object design (.rod) file in which the ReportType property setting is InformationObject, the formatting options are limited. To modify these fonts, you

set property values in the Properties page for the report component. You can modify the settings of the font properties that appear in Table 4-2.

Table 4-2 Font properties

Font property	Text affected
DataFont	Data that is retrieved from the data source
LabelFont	Display names for columns
PageDecorationFont	Page number and date in the footer
TitleFont	Title for displaying data rows

How to change the font properties for data rows



- 1 In Report Structure, right-click the report component and choose Properties.
- 2 In the Properties page, choose Expert Properties. Properties displays the advanced properties for the report component.
- 3 Expand Auto Contents, as shown in Figure 4-28.

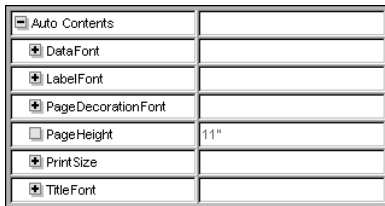


Figure 4-28 Auto Contents group of attributes

- 4 In Auto Contents, to access font properties, expand DataFont, LabelFont, PageDecorationFont, and TitleFont. Figure 4-29 shows the expanded LabelFont group.

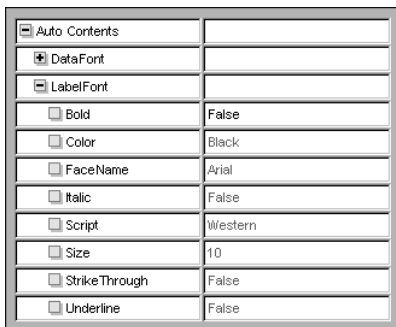


Figure 4-29 LabelFont group of attributes

- 5 Set the font attribute values.

- 6 Choose Report→Build and Run. If Requester appears, make your selections and choose OK. e.Report Designer Professional generates the data object executable file, runs it, and displays the data rows in Report Viewer. Figure 4-30 shows the result of setting Bold to True.

First Name	Last Name	Address	City	Ran	ID	Company	Phone
Betty	Murphy	18731 Douglas Av.	Allentown	A	182	CompuBoards	2155554753
Leslie	Taylor	16780 Pompton St.	Brickhave	B	211	InfoSpecialists Inc.	6175558428
Peter	Chandler	4030 Long Airport	Newark	B	118	Computer Systems Corp.	2015558624
Dan	Chandler	20789 Baden Av.	Brickhave	A	212	SigniEngineering Co.	6175559035
Dan	Firrelli	3764 Baden Av.	Newark	A	122	TeleMicroSystems	2015555171
Jose	Berajas	5420 Ingle Ln.	New	A	157	Computer Engineering	9145557064
Julie	King	25593 South Bay Ln.	Bridgewater	B	142	SigniSpecialists	2035552570
Wai Chung	Tam	613 Furth Circle	New	B	156	Advanced Design Corp.	9145556707
Valarie	Nelson	5677 Strong St.	Bridgewater	A	141	Design Design	2035551450
Diane	Chandler	23727 Kingston Rd.	New	B	204	Computer MicroSystems Corp.	5085557307
Kyung	Tseng	4656 Baden Av.	Cambridge	A	199	Technical Specialists Corp.	6175555910
Kwai	Fong	2732 Spinnaker Dr.	Albany	A	166	Advanced Solutions Inc.	5185559644
Stan	Smith	2399 Park Ln.	Sneadon'	A	161	Exosoft Corp.	9145557172
Maria	Avila	7497 Baden Av.	White	A	128	Computer MicroSystems Corp.	9145558205
Julie	Brown	7734 Strong St.	Trenton	A	148	Design Systems Inc.	6095551386

Figure 4-30 Report Viewer, showing the labels in bold font

5

Accessing an Actuate Information Object

This chapter contains the following topics:

- About information objects
- Setting up the report design to access information objects
- Modifying font attributes for information object query output in Actuate Query

About information objects

An information object is an encapsulated SQL query that you use as a basis for creating queries in report designers such as Actuate e.Report Designer Professional, Actuate e.Spreadsheet Designer, and BIRT Report Designer Professional. Other Actuate products also can use information objects. Information objects enable access to data from diverse data sources, such as database tables and views, other information objects, and result sets from stored procedures and open data access (ODA) data source queries. The encapsulation of these queries supports efficient report development by:

- Providing reusability across multiple reports and Actuate products. Report developers can use information objects to create reports in a report designer. Each report can use different columns, sorting rules, parameters, and filters, but the reports are based on the same data.
- Hiding the complexity of data access. Using information objects separates the data access logic from the report development process. Report developers can create queries without code, even for complex data from multiple sources. Information objects enhance report development by making it possible to create and modify queries without connecting to a data source or having knowledge of the data.

When you create a query in a report designer that is based on an information object, you can use all the data in the information object as your data set, or you can use a subset of the data in the information object. You also can join multiple information objects.

Information objects are created in Actuate Information Object Designer and stored in an Actuate iServer Encyclopedia volume. To work with an information object, you must have an account on an Encyclopedia volume with the privileges that are required to access and run the information object.

The information and procedures in this chapter apply to maps and information objects. A map is created in Actuate Information Object Designer and represents one of the following items:

- A database table
- A database view
- A query that is written in the database's native SQL
- A result set from a stored procedure
- A result set from an ODA data source query

For more information about maps, see *Designing BIRT Information Objects*.

Working with an information object

To use an information object with a report designer you complete the following tasks:

- Connecting to an Encyclopedia volume
- Creating a query that is based on one or more information objects
- Setting up the report design to display the data
- Running the report to view the data

Preparing to access information object data

Before you begin to create an information object query, you need the following information:

- The name of the Actuate iServer and the Encyclopedia volume that contains the information object
- A user name and password for the volume
- The name of the information object and its location in the Encyclopedia volume

You also need the following privileges that are sufficient to use the information object:

- Read privilege on the information object to create a query using the information object
- Execute or trusted execute privilege on the information object to run the query

Setting up the report design to access information objects

To access an information object, the report design must contain two components, an information object connection component and an information object data source component. You can create both types of components manually or by using Quick Report Wizard or Listing Report Wizard. In the wizards, you:

- Choose Actuate Data Integration Service Connection as the type of database.
- Choose Actuate Data Integration Service Data Source as the type of data stream.

You can use the user name and password defined in the connection while designing the report. Users running the report provide their own authentication information. You enable this functionality by using the `UseLoggedInUserCredentialsOnServer` property when you define the connection

properties for an information object. To instruct Actuate iServer to use the credentials of the user who runs the report to authenticate access to the Encyclopedia volume and information objects, set this property to False. To use the user name and password that are specified in the report design for all users who run the report, set this property to False. The default value is False.

How to define a connection for an information object manually

- 1 In Report Structure, ensure that you have an empty data source slot with an empty connection slot. If necessary, delete the existing data source and connection components.
- 2 In the toolbox, select Data to view the data tools, as shown in Figure 5-1.

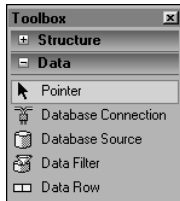


Figure 5-1 Viewing the data tools

- 3 Drag a database connection component from Toolbox—Data and drop it in an empty connection slot, as shown in Figure 5-2.

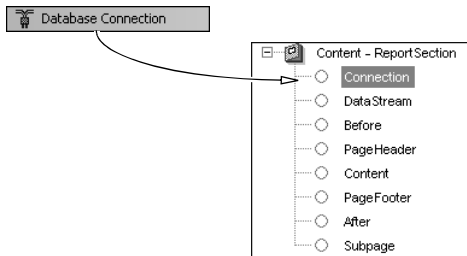


Figure 5-2 Dropping a database connection component in a Connection slot

- 4 In Select Component, select Actuate Data Integration Service Connection, as shown in Figure 5-3.

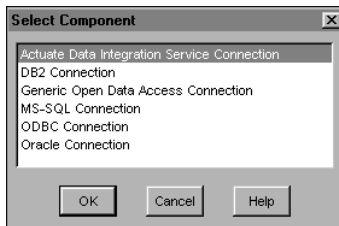


Figure 5-3 Selecting the type of connection for information objects

If you have already chosen a data source component, the choices on this dialog are limited to connection types that are compatible with your data source component.

Choose OK.

- 5 Drag a database source component from Toolbox—Data and drop it in an empty DataStream slot, as shown in Figure 5-4.

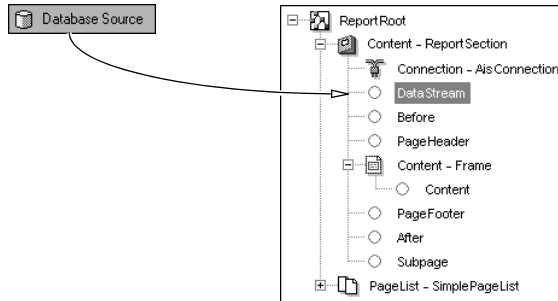


Figure 5-4 Dropping a database source component in a DataStream slot

- 6 On Select Component, select Actuate Data Integration Service Data Source, as shown in Figure 5-5.

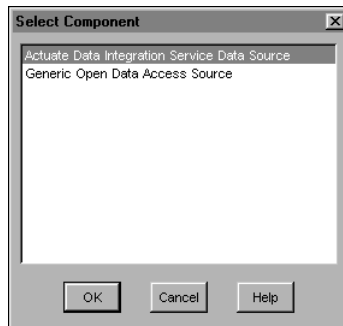


Figure 5-5 Selecting the type of data source used for information objects

Choose OK.

The information object data source appears in the report design, as shown in Figure 5-6.

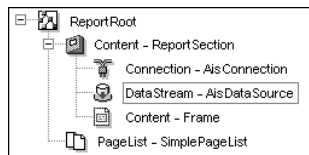


Figure 5-6 A report design with an information object data source

7 Optionally, set connection properties.

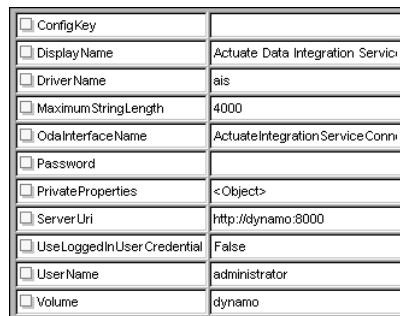
To set properties for the connection component:

- 1 In Report Structure, right-click the information object connection component. In the context menu, choose Properties.
- 2 In the Properties pane for the component, type values for the properties that appear in Table 5-1.

Table 5-1 Properties for a data source accessing information objects

Property	Description
Password	The password for the Encyclopedia volume account that has access to the information object.
ServerUri	The URL to the Actuate iServer computer that contains the Encyclopedia volume that stores the information objects. For example: <code>unidos.actuate.com</code>
UserName	The name of the Encyclopedia volume account that has access to the information object.
Volume	The Encyclopedia volume that stores the information object.

After you type values for these properties, the Properties page looks like the one in Figure 5-7.



☐ ConfigKey	
☐ DisplayName	Actuate Data Integration Service
☐ DriverName	ais
☐ MaximumStringLength	4000
☐ OdaInterfaceName	ActuateIntegrationServiceConn
☐ Password	
☐ PrivateProperties	<Object>
☐ ServerUri	http://dynamo:8000
☐ UseLoggedInUserCredential	False
☐ UserName	administrator
☐ Volume	dynamo

Figure 5-7 Connection properties for accessing information objects

You can now define a query by accessing Information Object Query Builder.

Modifying font attributes for information object query output in Actuate Query

If you plan to use Actuate Query to query information object (.iob) files, you can use e.Report Designer Professional to modify fonts used in Actuate Query output. Although you make this modification using e.Report Designer Professional, it does not affect any information object queries that were created in e.Report Designer Professional's Information Object Query Builder.

To display the Actuate Query output, an IOB uses the font settings that are specified in a template file that is located in an Encyclopedia volume. To modify these fonts using e.Report Designer Professional, you set property values for the report component of the template file, which is a report object design (.rod) file. You can change the settings of the font properties that appear in Table 5-2.

Table 5-2 Modifiable font properties

Font property	Text affected
DataFont	Data that is retrieved from the data sources
LabelFont	Column headings
PageDecorationFont	Page number and date in the footer
TitleFont	Title for Actuate Query output

After you set the values of the font properties in the template file, you complete the following tasks to make the file available to Actuate iServer System:

- Build a data object executable (.dox) file from the report object design (.rod) file.
- Publish the DOX to an Encyclopedia volume.
- Set the volume's Actuate Query Generation property to specify the path to, and name of, the DOX in the Encyclopedia volume.

For more information about using a DOX as a font template file for Actuate Query output, see *Configuring BIRT iServer*.

How to modify fonts used in Actuate Query output

This procedure explains how to modify fonts in a report object design (.rod) file used in Actuate Query output. You build a data object executable (.dox) file and publish it to an Encyclopedia volume.

- 1 In e.Report Designer Professional, open <eRDPro_HOME>\lib\AQTemplate.rod. AQTemplate.rod appears in Design Editor.
- 2 Save the file with a different name, such as AQTemplate_fonts.rod.



- 3 In Report Structure, right-click the report component, `ActuateQueryTemplate`. Choose Properties. The Properties page for the component appears.
- 4 On the Properties page, complete the following tasks:
 - 1 Expand Auto Contents, as shown in Figure 5-8.

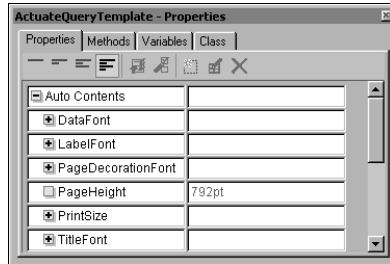


Figure 5-8 Auto Contents group of attributes

- 2 To set font properties in the Auto Contents group, expand the appropriate property:
 - ❑ DataFont
 - ❑ LabelFont
 - ❑ PageDecorationFont
 - ❑ TitleFont
- 3 Set the font property values.
- 4 Verify that the ReportType property setting is `ActuateQueryTemplate`.
- 5 Choose Report→Build. e.Report Designer Professional builds the data object executable (.dox) file.
- 6 Publish the DOX to an Encyclopedia volume:
 - 1 Choose File→Publish to Server. Publish and Preview Options appears.
 - 2 In Publish options, select Publish only.
 - 3 Provide additional information to specify to which Encyclopedia volume to publish the file.
- 7 Choose OK.

After you publish the DOX to an Encyclopedia volume, the system administrator must log in to the Actuate iServer's System Administration console to set the volume's Actuate Query Generation property. The Actuate Query Generation property value is the location and name of the DOX.

Accessing data in a comma-separated values (CSV) text file

This chapter contains the following topics:

- Accessing data from a CSV text file
- Creating a custom data source
- Creating variables to identify which text file to use
- Defining data row variables
- Displaying text file data in a report design
- Specifying processing of the data
- Running and viewing the report

Accessing data from a CSV text file

This chapter describes how to access CSV data from a text file. The CSV text file also can have commas embedded in a data value if the data value is enclosed by double quotes in the text file.

To retrieve CSV data from a text file, complete the following tasks in this order:

- Create a custom data source.
- Create variables to identify which text file to use.
- Define data row variables for the fields in the data.
- Set up display of the data in your report design.
- Specify how to process the data.
- Run and view the report.

Creating a custom data source

Create the custom data source by changing the superclass of the data stream to `AcDataSource` from the Actuate Foundation Classes library. For more information about AFC classes, see *Programming with Actuate Foundation Classes*.

How to create a custom data source

- 1 In Report Structure, right-click the `DataStream` component, and choose Properties.
- 2 Choose Class.
- 3 On Class, in Super class, as shown in Figure 6-1, type:

`AcDataSource`

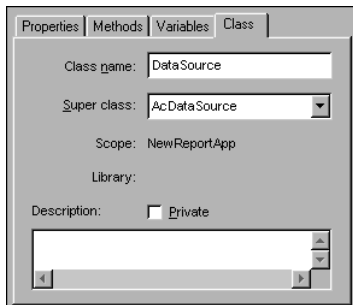


Figure 6-1 Creating a custom data source

This action changes the data stream to a custom data source.

Creating variables to identify which text file to use

e.Report Designer Professional requires the following information to connect to a text file:

- The file number that the operating system uses to identify the text file
- The name of the text file

To provide the data stream with this information, create variables to hold these values.

How to define a custom data source for your text file

In this example, the channel variable holds the file number and the DataInputFile variable holds the file name.

- 1 In Report Structure, select DataStream—DataSource.
- 2 In Properties, choose Variables.
- 3 Choose New.
- 4 On Class Variable, define Channel as shown in Figure 6-2, using the following values, then choose OK.



The screenshot shows the 'Class Variable' dialog box with the following settings:

- Name:** Channel
- Type:** Integer (with a 'Browse...' button)
- Externally defined data type:** ☐
- Storage:**
 - ☒ Instance (per object)
 - ☐ Static (shared by all objects)
- Visibility:** Public (with a 'Define...' button)
- Column:** (empty field)
- Buttons:** OK, Cancel, Help

Figure 6-2 Creating the Channel variable

- 5 On Class Variable, repeat steps 1 through 4, using the following values, to define DataInputFile, then choose OK:

- Name: DataInputFile
- Type: String
- Storage: Static

Selecting Static ensures that the variable is available to the entire report.

- Visibility: Parameter

You define DataInputFile as a parameter so users can enter a file name when they run the report.



- 6 In Variables, select DataInputFile. Choose Ellipsis.

- 7 On Parameter Properties, set the default value for DataInputFile to the full path name of your text file. For example, type:

C:\TextFile.txt

- 8 Select Required. Selecting Required ensures that the user must supply a value for DataInputFile.

Parameter Properties appears as shown in Figure 6-3.

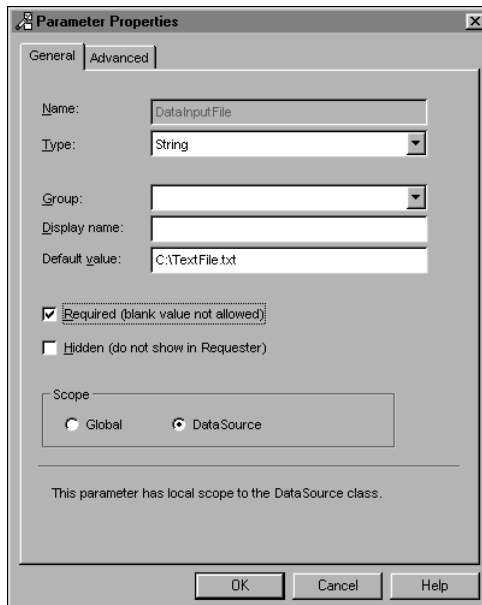


Figure 6-3 Parameter Properties

Choose OK.

Defining data row variables

For many types of data sources, e.Report Designer Professional defines variables for the data row automatically. For a CSV text file, you must define data row variables that correspond to the columns in your text file.

How to define data row variables for your text file

- 1 In Report Structure, expand DataStream—DataSource, and select DataRow.
The Properties page for the component appears.
- 2 Choose Variables.
- 3 Choose New.
Class Variable appears.
- 4 Specify the name and type of one of the fields in your text file. This variable's data type must correspond exactly to a field in the text file. Set Storage to Instance and Visibility to Public, as shown in Figure 6-4.

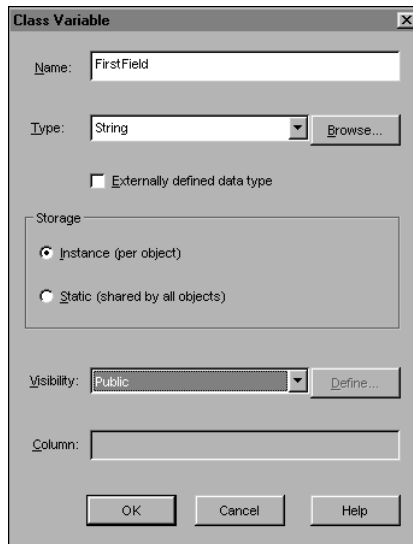


Figure 6-4 Creating the FirstField variable

Choose OK.

- 5 Repeat steps 1 through 4 for the other fields in your text file.

Figure 6-5 shows the settings for a text file that contains two string fields, FirstField and SecondField, and an integer field, Figures.

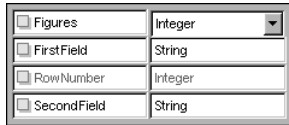


Figure 6-5 Variables for a text file that contains three fields

6 Choose View→Fields.

Fields appears as shown in Figure 6-6, displaying the variables that you defined.

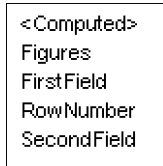


Figure 6-6 Fields, displaying the variables

Displaying text file data in a report design

After you create data row variables, you can lay out the report. The field list includes the data row variables that you defined. You select these variables and place them in your report design to display the text file data.

How to display text file data in a report design

- 1 In Layout, click in the content frame.
- 2 Drag the data row variables that you created from Fields, and drop them in the frame.
- 3 Resize and align the controls in the frame as needed.
- 4 Add any desired formatting for your report, such as text labels, dividing lines, and a report title.

For more information about laying out a report, see *Developing Reports using e.Report Designer Professional*.

Specifying processing of the data

To process CSV data from the text file, you override the `Start()`, `Fetch()`, and `Finish()` methods of the customized data stream component. These methods are inherited from its superclass, `AcDataSource`.

How to open a text file, retrieve rows, and close the file

This procedure adds code to open and close a text file. You could add error handling code to Function Start() As Boolean to handle an incorrect file name. This procedure also adds code to retrieve CSV data from a text file. Data values that are surrounded by double quotes can have embedded commas.

- 1 Right-click DataStream—DataSource, and choose Properties.
- 2 On the Properties page for the component, choose Methods.
- 3 On the Methods page for the component, double-click Function Start() As Boolean.

- 4 In DataSource::Start, modify Start() to open the text file.

Above the following code:

```
End Function
```

insert:

```
Channel = FreeFile( )  
Open DataInputFile For Input As Channel
```

- 5 In Properties—Methods, double-click Function Fetch() As AcDataRow.

- 6 In DataSource::Fetch, modify Fetch() to retrieve the rows.

Change the following code:

```
Set Fetch = Super::Fetch()
```

to:

```
Dim row As DataRow  
If EOF (Channel) Then  
    Exit Function  
End If  
Set row = New DataRow  
Input #Channel, row.FirstField, row.SecondField, row.Figures  
Set Fetch = row  
AddRow( Fetch )
```

- 7 In Properties—Methods, double-click Sub Finish().

- 8 In DataSource::Finish, modify Finish() to close the text file.

Above the following code:

```
End Sub
```

insert:

```
Close #Channel
```

Running and viewing the report

After you set up a report that uses data from a text file, you can run the report to access and display the data. To run the report, you choose Report→Build and Run. Requester displays a prompt for the data input file, as shown in Figure 6-7. If you do not change the value for the data file path name, the report accesses the default file.

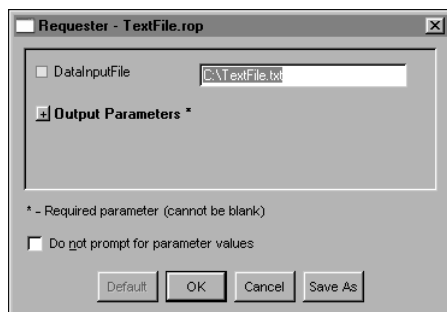


Figure 6-7 Prompting for the data input file name

The report output shows the data from the text file, as shown in Figure 6-8.

<i>Status Report</i>		
First Name	Last Name	# Figures Completed
Tom	Newman	4
Diana	Choi	7
Zina	Frei	12
Liz	Waller	17
Kala	William	21
Peter	Bertrand	5
Kate	Temple	11

Figure 6-8 Report output, showing the data from the text file

You can also add custom filters to sort the data or limit which rows appear.

Part Two

**Accessing data in a database or
ODBC data source**

Connecting to a database or ODBC data source

This chapter contains the following topics:

- About database and ODBC connections
- Preparing to access data using a database or ODBC driver
- Defining a database or ODBC connection
- Creating a query data source
- About AFC support for database and ODBC connections and queries

About database and ODBC connections

To access data from an ODBC data source, you use a query. To access data from a database, you use a SQL query or a stored procedure. This chapter describes how to connect to a database or ODBC data source and retrieve data using a query.

To connect to a database or ODBC data source, perform the following steps in this order:

- Gather the necessary information and set up access to the data source.
- Define a database or ODBC connection.
- Create a query data source.
- If necessary, access data using a stored procedure.

Preparing to access data using a database or ODBC driver

To access your database or ODBC data source, you must know the table and column structure of your data. You also need to know the joins that apply to your data source. If you do not know this information, obtain a database schema diagram from your data source administrator. For more information about joins and schemas, refer to the documentation for your database or ODBC data source.

You must ensure that your computer has access to the database or ODBC data source. How you access your data source determines what information you must provide to e.Report Designer Professional to create the connection.

Using an ODBC driver

Open Database Connectivity (ODBC) is an interface that allows an application to access data in any ODBC-compliant RDBMS. Since ODBC is such a mature technology, it is the main data access mechanism for Actuate reports. If your reports do not use ODBC connectivity yet, then you should plan to migrate your reports to use ODBC.

Preparing to access data using an ODBC driver

In addition to knowing your data structure, to access your data source using an Open Database Connectivity (ODBC) database driver, you must know:

- The ODBC data source name. Before you use an ODBC connection, you must create an ODBC data source using the ODBC administrator utility that comes with your ODBC driver software. For information about how to set up the

data source, see your ODBC documentation. The data source name can be a user data source name (DSN), a system DSN, or a file DSN.

- The user name and password, if your ODBC driver requires this information.
- The connection string for your ODBC data source. The connection string includes any other information the connection requires. The contents of this string depend on which driver you use with your ODBC data source. For example, the connection string can include the name of a Microsoft Excel file. If the data source name is a file DSN, the connection string is the name of the data source, preceded by FILEDSN=. For example, if the file DSN is MYDATA, the connection string you use in e.Report Designer Professional is: FILEDSN=MYDATA.

If you are using the ODBC driver included with e.Report Designer Professional to access an Oracle database and want to access NCHAR and NVARCHAR fields, add the following phrase to the connection string:

`EnableNcharSupport=1`

For information about the connection string for your database driver, see your ODBC driver documentation.

Configuring an ODBC driver

The e.Report Designer Professional installation configures several sample ODBC data sources as user DSNs. If you are not the user who installed e.Report Designer Professional or you want to access an additional ODBC data source, you must configure the ODBC data source.

A localized e.Report Designer Professional installation does not install and configure a localized database. To use a localized database, such as a Japanese version of sfdata.mdb, you must configure the localized database as an ODBC data source.

If you use an ODBC driver and want to use parameters in the SELECT statement, SQL-92 imposes limitations on where you can place a parameter in the SELECT statement. For more information about these limitations, see:

http://msdn.microsoft.com/library/default.asp?usr=/library/en-us/odbc/htm/odbcstatement_parameters.asp

How to configure an ODBC data source

With ODBC data, users who have the same drivers installed can share file DSNs. To make a data source accessible to all users on a computer, configure it as a system DSN. The following procedure explains how to configure an ODBC data source as a system DSN on Microsoft Windows. For more information about configuring an ODBC data source, see your operating system documentation.

- 1 Depending on your operating system, complete one of the following tasks:

- In Windows 7 Professional, navigate to and double-click the following file:
`\Windows\SysWOW64\odbcad32.exe`
- In Microsoft Windows Server 2003, Windows Vista, or Windows XP Professional, in Control Panel, choose Administrative Tools→Data Sources (ODBC).

2 Select System DSN.

ODBC Data Source Administrator—System DSN displays a list of system data sources as shown in Figure 7-1. The list includes system data sources that were configured during the e.Report Designer Professional installation.

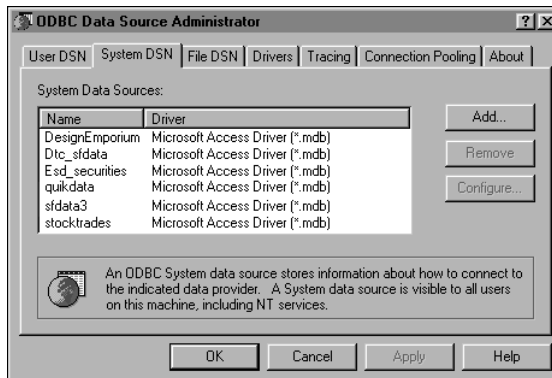


Figure 7-1 List of system data sources

3 Choose Add.

Create New Data Source appears as shown in Figure 7-2, displaying a list of ODBC drivers that are installed and available.

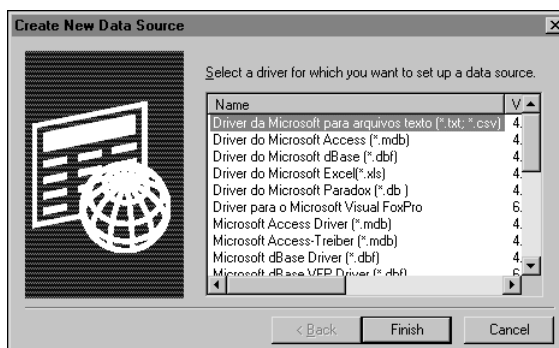


Figure 7-2 List of ODBC drivers

- 4 Select Microsoft Access Driver (*.mdb), and then choose Finish.
- 5 On ODBC Microsoft Access Setup, specify data source information:

- 1 In Data Source Name, type the name of the data source. Optionally, type a description.

ODBC Microsoft Access Setup looks like the one in Figure 7-3.

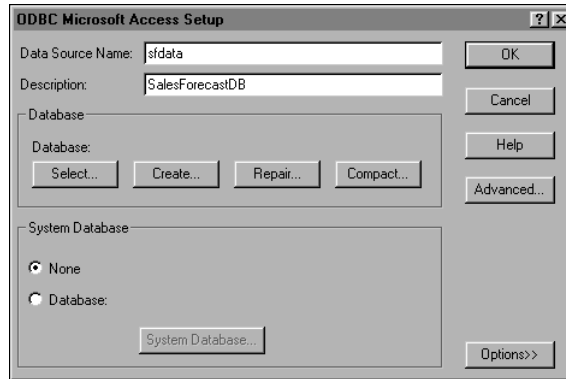


Figure 7-3 ODBC Microsoft Access Setup

- 2 In Database, choose Select.
- 3 On Select Database, in Directories, navigate to the directory that contains the data source, as shown in Figure 7-4.

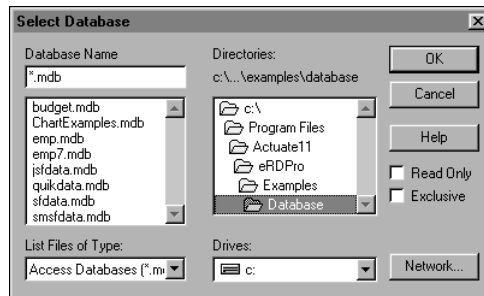


Figure 7-4 Navigating to the directory with the data source

- 4 In the list that appears below Database Name, select the data source. Choose OK.
- 6 On ODBC Microsoft Access Setup, choose OK.
- 7 On ODBC Data Source Administrator—System DSN, choose OK.

Using a native database driver

Actuate e.Report Designer Professional and Actuate iServer have integrated ODBC drivers that provide an out-of-the-box connectivity mechanism for Oracle, DB2, and MS SQL Server. Actuate recommends that you use those ODBC drivers for your reports. The driver that you use depends on the database version. For

example, if you have SQL Server 2005, you use the ActuateDD 6.0 SQL Server Wire Protocol driver. If you have SQL Server 2008, you use the ActuateDD 6.0 SQL Server Native Wire Protocol driver.

Some native database connection types, such as Oracle, require the installation of a database client before you can connect to the database server. In that case, you must also define the database connection on the server before you can successfully deploy a report object executable (.rox) file. For more information about defining an Actuate iServer database connection, see *Configuring BIRT iServer*.

The database driver that you use determines the type of information that you must provide to e.Report Designer Professional. For details about a particular connection type, see the documentation for the database. Table 7-1 shows the information that you must provide to e.Report Designer Professional and any optional information that you can provide for each type of database server.

Table 7-1 Required and optional connection information

Database server type	Required connection information	Optional connection information
DB2	■ Data source name ■ Password, if required for your user account ■ User Name	
SQL Server	■ Password, if required for your user account ■ User name	Server name. If not supplied, e.Report Designer Professional uses a locally defined server as the default server.
Oracle	■ Database interface, such as acorcl90 ■ Host string, such as oran9i ■ Password, if required for your user account ■ User name	

Defining a database or ODBC connection

To define a connection for a database or ODBC data source, choose Tools>Database Connection or perform the steps in the following procedure. If you do not supply the required connection property values by setting them for

the connection component, Database Login appears and requests the property values when e.Report Designer Professional connects to the data source.

How to define a connection

- 1 In Report Structure, ensure that a connection slot and its associated DataStream slot are empty, as shown in Figure 7-5. If necessary, right-click an existing connection or data source component and choose Delete.

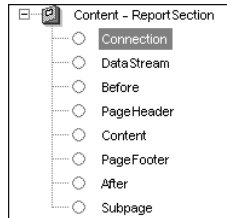


Figure 7-5 An empty connection slot and DataStream slot

- 2 Open the toolbox. Select Data to see the data tools, as shown in Figure 7-6.

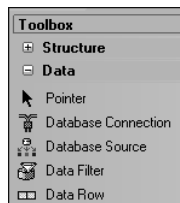


Figure 7-6 The data tools

- 3 Drag a database connection component from Toolbox—Data and drop it in Connection, as shown in Figure 7-7.

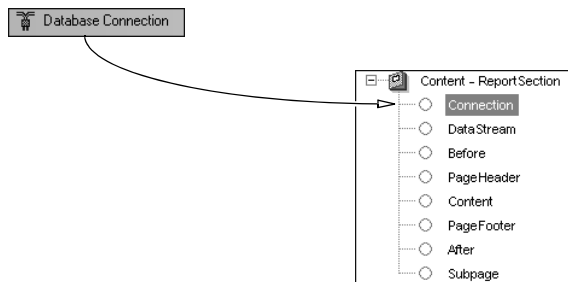


Figure 7-7 Adding a connection component

Select Component displays all connection types that are compatible with your data source component, if any. If you have not chosen a data source component, e.Report Designer Professional displays all possible connection types.

- 4 In Select Component, select the type of connection, as shown in Figure 7-8. For an ODBC connection, select ODBC Connection. For a native database driver, select the type of database.

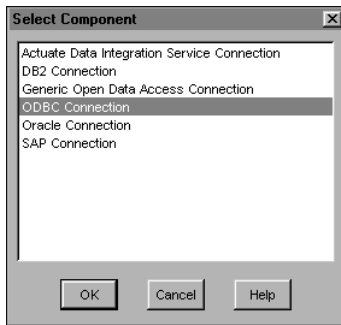


Figure 7-8 Selecting an ODBC connection

Choose OK.

- 5 In Component Properties, as shown in Figure 7-9, select the data source from the drop-down list, and type the values for the connection properties that your data source server requires.

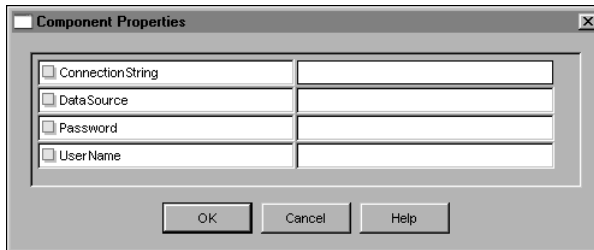


Figure 7-9 Component Properties

Alternatively, you can choose OK, right-click the connection component in Report Structure, and choose Properties. You can then set the connection property values on the Properties page for the component.

Creating a query data source

A query data stream always includes at least two components, a query data source and a data row, as shown in Figure 7-10. You see these components in Report Structure when you build a report.

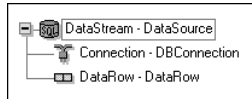


Figure 7-10 A query data stream with a data source and a data row

The query data source component contains a SQL query, which it uses to retrieve data from the data source. Before you create the query, the report design must have a query data source component.

If you start a new report design using a report wizard or a blank report, e.Report Designer Professional adds the data stream components to the report design, including a graphical query data source component.

You can use the graphical query data source component or delete it. If you create a query graphically and later convert it to a textual query, the conversion replaces the original graphical query data source component with a textual one.

To add query data source and data row components to a report design, use the toolbox. You can also use an existing query data source or data row from a library or from another report design.

For more information about adding a component to a report, see *Developing Reports using e.Report Designer Professional*.

How to add query data source and data row components to a report design using the toolbox

In this procedure you use the toolbox to add a query data source and data row component to a report design.



- 1 Ensure that the report design contains a database connection. For more information about defining a connection, see “Defining a database or ODBC connection,” earlier in this chapter.



- 2 Drag a database source component from Toolbox—Data and drop it in the report section’s DataStream slot, as shown in Figure 7-11.

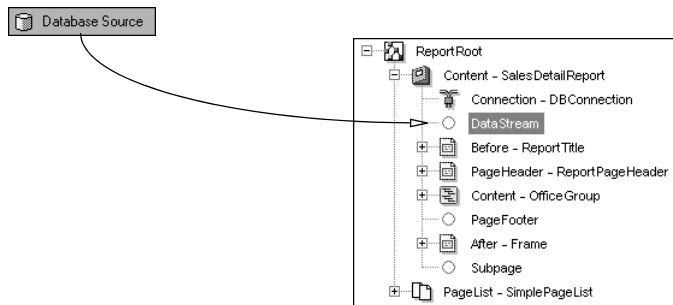


Figure 7-11 Adding a database source component

- 3 Use Select Component, as shown in Figure 7-14 to choose how to create your query:

- To create your query graphically, select SQL Query Data Source (Graphical). The data source appears in Report Structure and looks like the one in Figure 7-12.

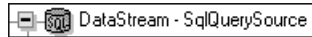


Figure 7-12 A graphical SQL query data source

- To create your query using SQL code, select SQL Query Data Source (Textual). The data source appears in Report Structure and looks like the one in Figure 7-13.

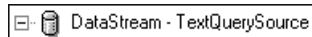


Figure 7-13 A textual SQL query data source

Choose OK.

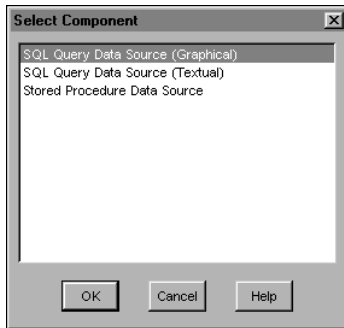


Figure 7-14 Selecting a data source type



- 4 Drag a data row component from Toolbox—Data and drop it in the query data source component's DataRow slot. Figure 7-15 shows the data row component with a graphical query data source.

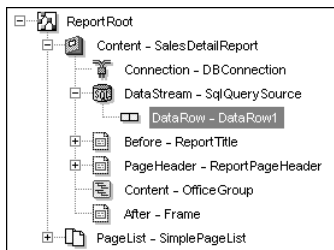


Figure 7-15 A graphical query data source with a data row component

About AFC support for database and ODBC connections and queries

The Actuate Foundation Class AcConnection establishes the protocol for connecting, disconnecting, and generating error messages if a connection fails. Database and ODBC connections all use classes that are derived from AcDBConnection, which derives from AcConnection.

Table 7-2 lists an ODBC data source and the databases to which you can connect. The table also lists the Actuate Foundation Classes that support connecting to each data source. These classes define variables, such as user name and password, that are necessary for establishing a connection. For more information about these classes, see *Programming with Actuate Foundation Classes*.

Table 7-2 Data sources and their corresponding connection classes

Data source	Connection class
DB2	AcDB2Connection
MicrosoftSQL	AcMSSQLConnection
ODBC	AcODBCConnection
Oracle	AcOracleConnection

To access data in a database or ODBC data source, you typically use a SQL query data source. The SQL query data source assembles the SQL SELECT statement that you created using the graphical or textual query editor and sends the statement to the database or ODBC data source.

To communicate with the database or ODBC data source, the data source uses statements (AcDBStatement) and cursors (AcDBCursor). A statement provides a way to execute a SQL SELECT statement. The result of a SELECT statement is typically a set of records. When a SQL SELECT statement returns table data, a cursor manages the retrieval of data rows. It also tracks the row position in the record set as the database or ODBC data source sends each row to the data source.

As shown in Figure 7-16, the report section, data source, connection, statement, and cursor interact in the following ways:

- A report section owns, instantiates, and deletes a data source. If the connection is in a report section, the report section owns the connection.
- The data source instantiates and deletes a statement or cursor. The data source calls the connection's Prepare() method to instantiate a statement. Then, the data source calls the statement's AllocateCursor() method to instantiate a cursor. If the connection is in the data source, the data source instantiates and deletes the connection.

- The data source uses the connection.
- The statement uses the connection.
- The cursor uses the statement.

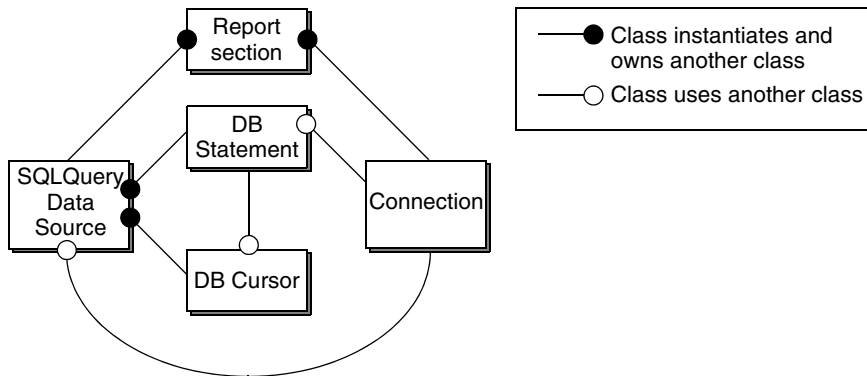


Figure 7-16 Interaction between the report section, data source, connection, statement, and cursor

For more information about Actuate Foundation classes, see *Programming with Actuate Foundation Classes*.

Creating a database or ODBC query

This chapter contains the following topics:

- About creating databases and ODBC queries
- Creating a query
- Creating a query graphically
- Creating a query by writing a SQL SELECT statement
- Converting a graphical query to a textual query
- Previewing the rows that a database or ODBC query returns
- Changing the properties of a result column
- Specifying sorting

About creating databases and ODBC queries

A query is a SQL SELECT statement that specifies the data to retrieve from a data source. To access data from an ODBC data source, you use a query. To access data from a database, you use a query or a stored procedure. This chapter describes how to create a query.

To access data from your database or ODBC data source, perform the following steps in this order:

- Gather the necessary information and set up access to the data source.
- Define a database or ODBC connection.
- Create a query data source.
- Create a query.

To support users specifying additional filter conditions for a report, you can provide parameters in your query.

To improve performance of an existing query, you can view the SQL that the query uses and tune it in your data source. If the tables or columns in your data source change, you must synchronize the query with the new table and column definitions in the data source.

Creating a query

When you use e.Report Designer Professional, you do not need a detailed understanding of the SQL language. You can use Query Editor to select tables, columns, and other query options. e.Report Designer Professional writes the SQL SELECT statements as you make selections in Query Editor. In most situations, you create SQL queries using Query Editor.

You also can write your own SQL SELECT statements using Textual Query Editor. Write your own SQL SELECT statements only if you want to use an existing SQL statement, if you prefer using SQL directly, or if you want to tune your SQL SELECT statement to obtain better performance from your data source. Textual Query Editor supports writing SQL SELECT statements, modifying them, and viewing related information. When you use Textual Query Editor, e.Report Designer Professional supports creating the data row and binding the columns. You also can override AFC methods to specify your SQL SELECT statement. For more information about writing SQL, refer to a SQL manual or the documentation for your database or ODBC data source.

Even if you are comfortable writing SQL code, consider creating a query graphically first. You later can convert a query that you define graphically to a textual query. You cannot convert a textual query to a graphical query.

A textual query and an equivalent graphical query return the same results when accessing most types of databases. Some databases, such as DB2, have characteristics that cause different results for queries in which a user provides values to filter the results of the query. Textual queries are filtered by Actuate iServer, while graphical queries push the condition into the WHERE clause and rely on the database to filter the results. String comparison in Actuate iServer is case-sensitive and does not ignore trailing spaces. String comparison on some databases such as DB2 is not case-sensitive and ignores trailing spaces. For example, a DB2 database accepts "ABC" as equal to "abc " while Actuate iServer considers them not equal. A similar issue occurs in Oracle databases, because Oracle pads the value in CHAR data type columns with blank spaces to fill the fixed length of a column. Actuate iServer does not consider "ABC" equivalent to "ABC ", but Oracle's string comparison for CHAR data types ignores trailing spaces.

To ignore trailing spaces in a textual query, use TRIM() or an equivalent SQL function that is understood by the database to trim the trailing spaces from values before comparing them. To replicate case-insensitivity in a textual query, use UPPER(), LOWER() or a similar SQL function that is understood by the database to convert the case of values before comparing them.

e.Report Designer Professional uses the sections in your report design to determine a default sort order. You can modify or override this default functionality.

You can change the default data type or display length for the columns that your query returns. You also can add a reference name if you plan to use the name of a column in customized code and want a simpler or clearer name.

Creating a query graphically

To create a query graphically, you perform the following procedures:

- Opening Query Editor and Database Browser
- Using tables, views, and synonyms
- Selecting and modifying columns and creating computed fields

You can also choose to perform any of the following optional procedures:

- Summarizing data from multiple rows
- Adding conditions to a query
- Viewing the generated SQL SELECT statement
- Previewing the rows that a database or ODBC query returns

- Changing the properties of a result column
- Specifying sorting

Opening Query Editor and Database Browser

You create graphical queries using Query Editor and select tables for the query using Database Browser. You can also see and select views, system tables, and synonyms in Database Browser, depending on which type of data source you use.

Database Browser shows a graphical representation of the tables and views that you can access through the connection. Synonyms in data sources provide alternate names for tables and views. When Database Browser displays synonyms, it displays the table or view name of the synonym.

How to open Query Editor and Database Browser

- 1 Ensure that the report design contains a connection and a graphical query data source.
- 2 Choose View→Data.

If you do not have an open connection, e.Report Designer Professional attempts to connect to the data source using the connection properties defined in the connection component. If e.Report Designer Professional is unable to connect, Database Login appears, as shown in Figure 8-1.

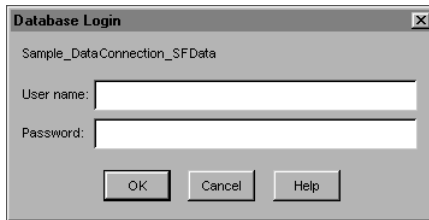


Figure 8-1 Database Login

- 3 In Database Login, type the required login credentials, then choose OK. In Query Editor the upper part is empty, and the lower part contains no values, as shown in Figure 8-2.

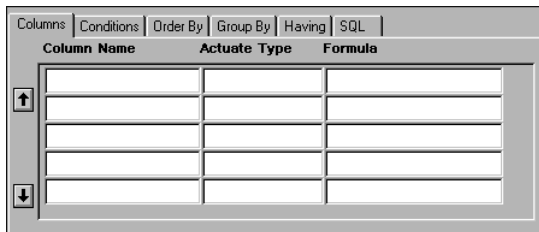


Figure 8-2 Lower part of Query Editor

Database Browser shows the available tables, views, and synonyms, as shown in Figure 8-3.

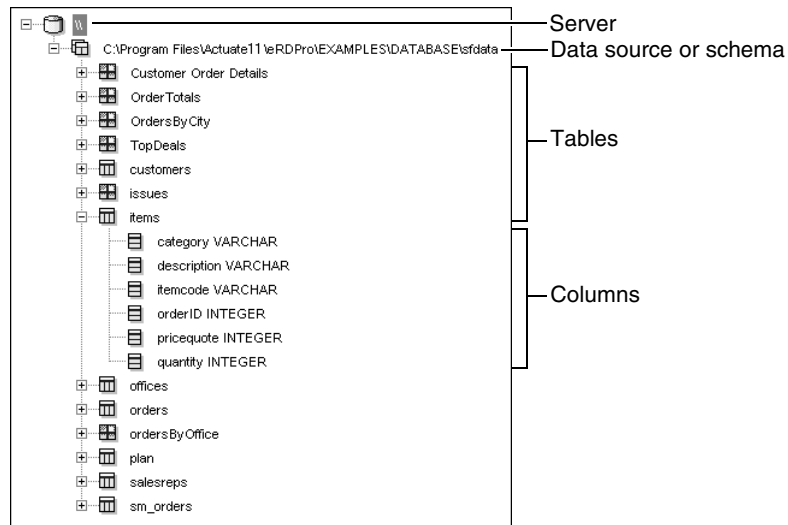


Figure 8-3 Available tables, views, and synonyms

If your data source supports a hierarchical naming structure, you can specify whether the data source and owner names appear as part of table names in Query Editor by choosing SQL→Column Qualifiers.

How to specify the display and content in Database Browser

This procedure describes how to use the Database Browsers.

- 1 Choose Tools→Options→Query Editor.
- 2 In Tables and Views, specify whether to display tables, views, or both tables and views in Query Editor.
 - Select Show tables and views to see both tables and views.
 - Select Show tables only to see only tables.
 - Select Show views only to see only views.
- 3 In Names, specify whether synonym names or the underlying base table or view names appear in Database Browser.
 - Select Base names to display only the underlying base table or view names for synonyms.
 - Select Synonyms to display only the synonym name.
 - Select Base names and synonyms to display both the synonym name and the base table or view name for the synonym.

- 4 Select Show system objects to display accessible data source objects that the user did not build.
- 5 In Owner pattern match, if the owner name must match a pattern, type a value. Use a wildcard character (*) to match any character.
- 6 In Table or view pattern match, if the table or view name must match a pattern, type a value. Use a wildcard character (*) to match any character.
- 7 In table or view name qualification, specify whether Database Browser displays synonym names or the underlying base table or view names.
 - Select table to display only the names of tables and views.
 - Select owner.table to display the names and owners of tables and views.
 - Select qualifier.owner.table to display the names, owners, and another qualifier for tables and views. The third qualifier is defined by the source database.

Using tables, views, and synonyms

After opening Query Editor, select the data source tables, views, and synonyms that provide the information for the report. For the rest of this chapter, the term tables also includes views and synonyms that are used in place of tables. As you choose tables, Actuate e.Report Designer Professional writes the FROM clause of the SQL SELECT statement.

You can use the same table multiple times. For example, you can use a table of address data to obtain both a Ship To address and a Bill To address in the same data row.

How to select a table

-  1 In Database Browser, expand nodes by choosing the plus icons until you see the table that you want to use.

If desired, you can preview the columns and data in the table before choosing to use this table.
- 2 Drag the table from Database Browser and drop it in the upper part of Query Editor, causing the table in the Query editor to appear as shown in Figure 8-4.

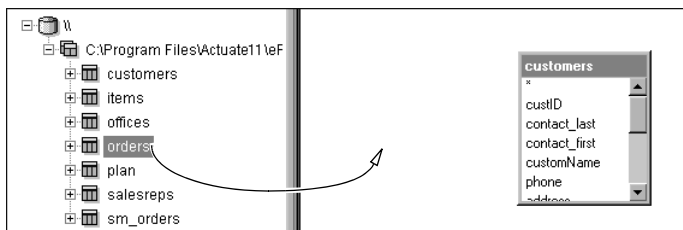


Figure 8-4 Adding a table

If you experience poor performance when you drop tables in Query Editor, or if your data source contains more than 100 tables, disable the automatic generation of joins. To disable the automatic generation of joins, choose SQL → Auto Joins.

- 3 If you drag and drop a table that is already in Query Editor, Alias Name appears, as shown in Figure 8-5.



Figure 8-5 Alias Name, showing a default alias

- 4 To assign the default alias name, choose OK. Alternatively, if you do not want to use the default alias, type an alternate name for the table before choosing OK.

How to preview the columns and data in a table

- 1 Right-click the name of a table in Database Browser or in the upper part of Query Editor, and choose Preview Table Data. The Preview Table Data, as shown in Figure 8-6, has a status bar on the bottom left that lists performance statistics. For long-running queries, you can use these statistics to help tune your query.

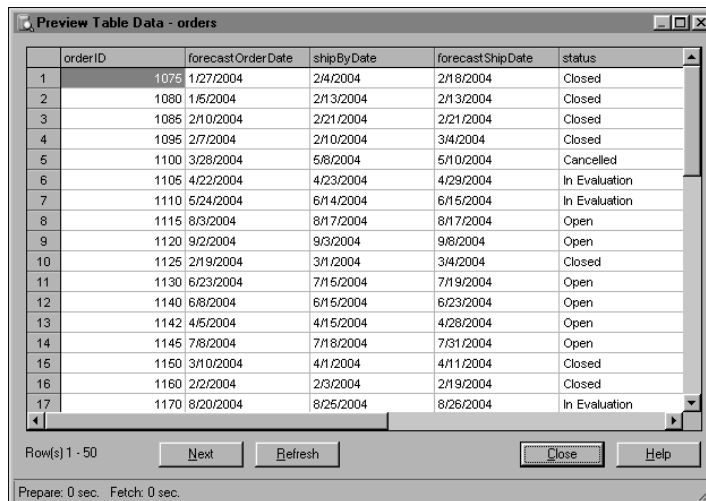


Figure 8-6 Previewing the columns and data in a table

- 2 Expand the width of any truncated columns by clicking on the right border of the column label.

- 3 To keep the first column or first several columns visible while you scroll through the rest of the data, select the left margin of the first column's label, and drag it to the last column that is to remain visible. Figure 8-7 shows the results of freezing the first two columns then scrolling horizontally.

orderID	forecast Order Date	status	issue	askByDate
1	1075 1/27/2004	Closed	<null>	2/22/2004
2	1080 1/5/2004	Closed	<null>	1/12/2004
3	1085 2/10/2004	Closed	<null>	3/27/2004
4	1095 2/7/2004	Closed	<null>	4/7/2004
5	1100 3/28/2004	Cancelled	Can we renegotiate...	5/13/2004
6	1105 4/22/2004	In Evaluation	They decided to go...	4/23/2004
7	1110 5/24/2004	In Evaluation	<null>	6/4/2004
8	1115 8/3/2004	Open	<null>	8/9/2004
9	1120 9/2/2004	Open	<null>	9/7/2004
10	1125 2/19/2004	Closed	<null>	3/17/2004
11	1130 6/23/2004	Open	<null>	7/3/2004
12	1140 6/8/2004	Open	<null>	6/14/2004
13	1142 4/5/2004	Open	They haven't decid...	4/7/2004
14	1145 7/8/2004	Open	This is the last orde...	7/10/2004
15	1160 3/10/2004	Closed	We need to keep in...	4/3/2004
16	1160 2/2/2004	Closed	<null>	4/17/2004
17	1170 8/20/2004	In Evaluation	<null>	8/23/2004

Figure 8-7 Freezing the first two columns

- 4 To keep the first row or first several rows visible while you scroll through the rest of the data, select the top margin of the first row, and drag it to the last row that is to remain visible. Figure 8-8 shows the results of freezing the first seven rows then scrolling vertically.

orderID	forecast Order Date	status	issue	askByDate
1	1075 1/27/2004	Closed	<null>	2/22/2004
2	1080 1/5/2004	Closed	<null>	1/12/2004
3	1085 2/10/2004	Closed	<null>	3/27/2004
4	1095 2/7/2004	Closed	<null>	4/7/2004
5	1100 3/28/2004	Cancelled	Can we renegotiate...	5/13/2004
6	1105 4/22/2004	In Evaluation	They decided to go...	4/23/2004
7	1110 5/24/2004	In Evaluation	<null>	6/4/2004
26	1235 7/23/2004	Selected	There is a shift in p...	7/27/2004
27	1240 2/17/2004	Closed	<null>	2/18/2004
28	1245 9/5/2004	In Evaluation	This has moved to t...	9/9/2004
29	1260 6/29/2004	In Evaluation	<null>	6/14/2004
30	1261 6/1/2004	In Evaluation	<null>	6/17/2004
31	1265 1/28/2004	Closed	<null>	4/7/2004
32	1275 7/17/2004	In Evaluation	<null>	7/23/2004
33	1280 8/16/2004	In Evaluation	Can we deliver the ...	8/24/2004
34	1285 7/30/2004	Open	<null>	8/5/2004
35	1290 7/9/2004	In Evaluation	<null>	7/18/2004

Figure 8-8 Freezing the first seven rows

To delete a table from the upper pane of Query Editor, select the table, and press Delete.

How to change the format and names of the tables

After you add a table to a query, you can indicate how table references appear in the SQL code that Actuate e.Report Designer Professional generates.

- 1 In the upper pane of Query Editor, double-click the table.

Query Editor—Table Property appears, as shown in Figure 8-9.

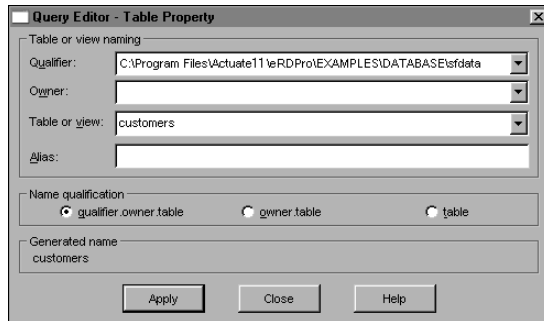


Figure 8-9 The table properties

- 2 Modify the following table properties as desired:
 - In Alias, type a name to use for the table or view.
 - In Name qualification, select the desired format for the table or view name.
- 3 Choose Close.

Creating table joins

A join is a SQL query operation that uses the values in the join fields to match records in different tables. A join ensures that queries correctly associate the fields within different tables.

When you add more than one table to a SQL query, e.Report Designer Professional creates joins to link related tables if it can determine the columns on which to join the tables. When you create a query, view the joins that e.Report Designer Professional creates for your data source, and ensure that they match the structure of your data source schema. As you create or delete joins, Actuate e.Report Designer Professional changes the WHERE clause of the SQL SELECT statement for your query.

About the automatic generation of joins

e.Report Designer Professional applies the following criteria in this order when generating joins:

- Any relationship information, or metadata, that is available in the data dictionary of a data source. For example, a data dictionary can contain the information that one of the tables has a foreign key that maps to the other table's primary key.
- Matching column names and types for an ODBC driver that does not support relationship metadata. For example, e.Report Designer Professional matches column names and types to create automatic joins for the sample Sfdata.mdb Microsoft Access database.

Creating and deleting joins manually

The type of join you can create depends on the type of data source. Table 8-1 describes the types of joins that Query Editor supports.

Table 8-1 Types of joins that Query Editor supports

Type of join	Returns
Inner join	Only records from two tables where the values of the joined fields have a specified relationship, such as equal to or greater than
Left outer join	All records from the table on the left side of the join even if the other table does not have a record with the specified relationship, such as equal to or greater than
Right outer join	All records from the table on the right side of the join even if the other table does not have a record with the specified relationship, such as equal to or greater than

How to create a join manually

- 1 In Query Editor, drag the primary key column name from a table and drop it on the foreign key column name in another table, as shown in Figure 8-10.

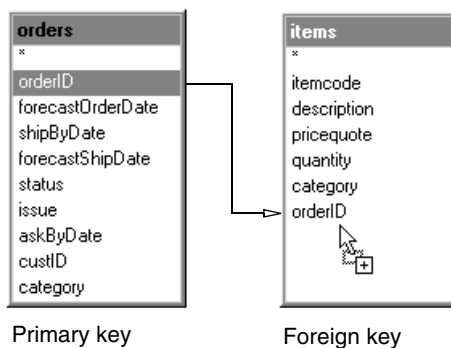


Figure 8-10 Creating a join manually

If your primary key requires more than one column to establish a unique value, repeat step 1 to drop additional column names on the same foreign key. For example, in a data source of parts from several manufacturers, the part ID alone is not necessarily unique. Several manufacturers can use the same number. To establish a unique key in these cases, use both the part ID and the manufacturer name.

2 Change the type of join, if desired:

- 1 Double-click the join operator icon, shown in Figure 8-11.

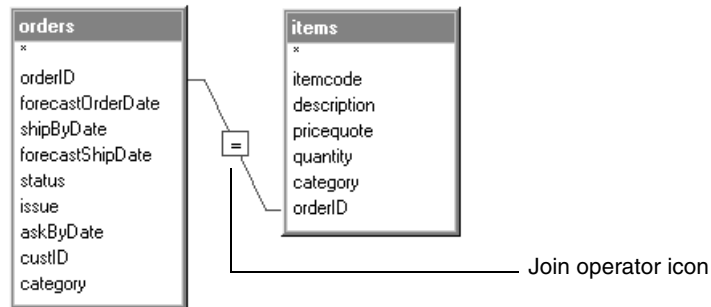


Figure 8-11 Join operator icon

Join Property appears.

- 2 On Join Property, select an operator from the drop-down list, as shown in Figure 8-12.

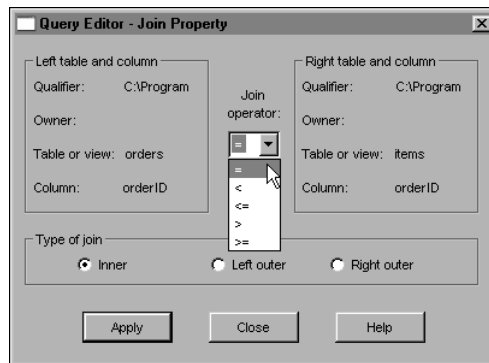


Figure 8-12 Selecting a join operator

- 3 In Type of join, select one of the options:

- ☐ Inner
- ☐ Left outer

- ❑ Right outer

Choose Close.

How to delete a join

Select the join line or join operator, as shown in Figure 8-13, and press Delete.

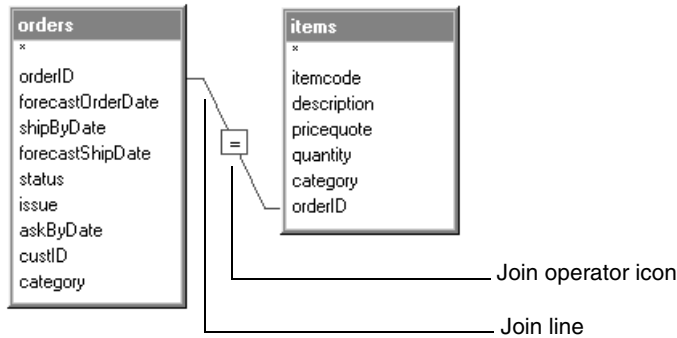


Figure 8-13 Join operator and join line

Modifying the generated FROM clause for a query

After you select tables for a query, e.Report Designer Professional generates a FROM clause for the query to request data from those tables. It also refers to these tables in other clauses of the SQL SELECT statement. For example, after you choose columns from a table, e.Report Designer Professional creates a SELECT clause that identifies each column in table.column format.

If you know SQL, you can replace the automatically generated FROM clause with your own FROM clause. Typically, report developers and data architects use the default FROM clause. If you choose to customize the FROM clause, you can use several techniques:

- Customize the FROM clause in the query editor.
This section explains this technique. You can change the format and names that identify tables or edit the entire FROM clause for the SQL SELECT statement.
- Convert the graphical query to a textual query, and change the SQL code.
- Override the ObtainSelectStatement method.
For more information about the ObtainSelectStatement method, see *Programming with Actuate Foundation Classes*. For an example of overriding the ObtainSelectStatement, see “Viewing the SQL SELECT statement that e.Report Designer Professional sends to the data source” in Chapter 10, “Maintaining a database or ODBC query.”

How to edit or replace the FROM clause for a graphical query

- 1 In Query Editor, choose SQL→Custom From Clause.
- 2 On Custom FROM Clause, select Use custom FROM clause. In the available field, type the FROM clause.

In Figure 8-14, two static parameters, `:[OrderHeader.DateOrdered]` and `:[OrderHeader.DateShipped]` support users specifying values when they run the report.

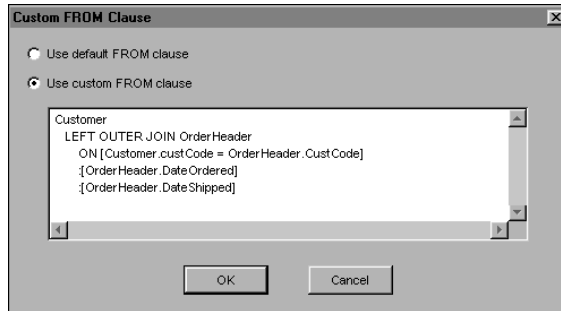


Figure 8-14 Two static parameters

Choose OK.

Selecting and modifying columns and creating computed fields

You can write a query to select all columns or only some of the columns from each table. You also can create a computed field. As you select columns, e.Report Designer Professional writes the SELECT clause of the SQL SELECT statement and declares variables in the data row. After you select or create columns, you can change the order of the columns in the data row.

Selecting columns from a table

You select columns to include in a query using Query Editor. The following procedure describes how to select columns using Query Editor.

How to select columns for a query

- 1 In Query Editor, choose Columns.
Query Editor—Columns appears.
- 2 Select columns using one of the following methods:
 - In Column Name, use the drop-down list to select a column.
 - Drag the column from the upper part of Query Editor and drop it in Column Name.

You can select and drag multiple columns. To select all columns, drag the asterisk (*), as shown in Figure 8-15. This technique selects all columns that are in the table when the report object executable (.rox) file runs, which can differ from the columns that now appear in Query Editor.

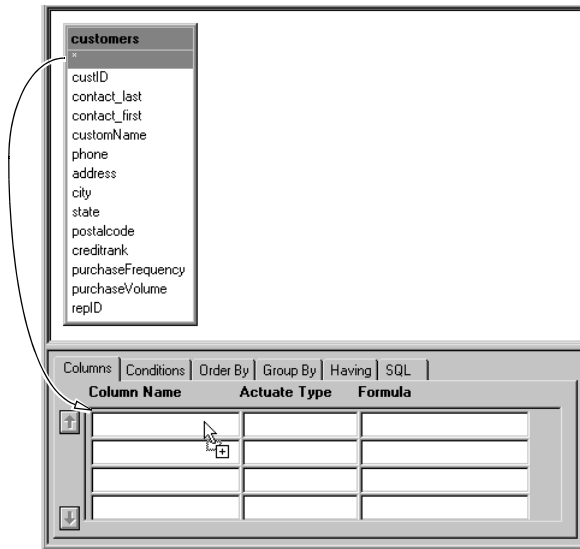


Figure 8-15 Selecting all columns

How to specify that the query only returns distinct rows

In some cases, a query can return multiple rows that have the same values. For example, a query that returns only the customer name and phone number from a table that contains order information will have duplicate rows if a customer ordered more than once. To display only unique rows, you can specify that the query returns only rows that have distinct values.

To display only rows that have distinct values, in Query Editor, choose SQL→Distinct.

How to view the data in a column

Right-click the name of a column in Database Browser or in the upper part of Query Editor, and choose Preview Column Data.

Preview Data appears, as shown in Figure 8-16.

How to delete a column

To delete a column using Query Editor, choose Columns, select the column name, and press Shift+Delete.

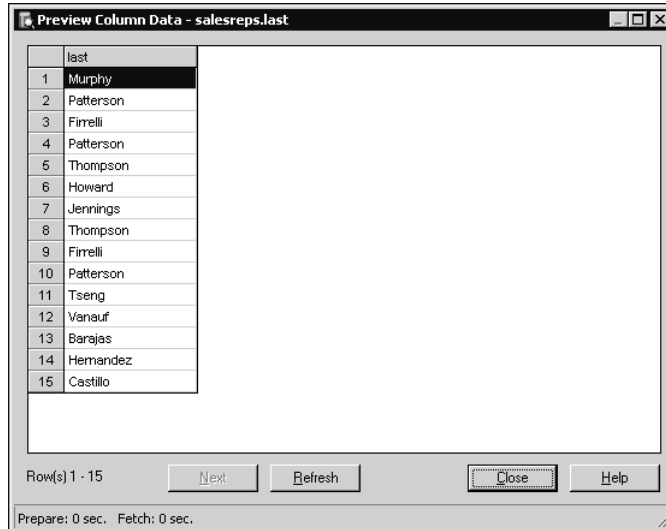


Figure 8-16 Viewing the data in a column

Creating a computed field

Typically a field's value is the value of a field in the data source. A computed field is a field whose value is computed from an expression, often involving one or more fields in the data source. For example, if you have fields in the data source that provide the type of item, the number of those items that were ordered, and the cost per item, you could use a computed field to calculate the total cost for that type of item.

You can create a computed field as a control, as a data row variable, or in a query editor. This section describes how to create a computed field as part of the data source query.

You can use a formula on one or more columns to specify a computed field. For example, to return the extended price for an item in a sales order, you might use the following computation:

```
[items.price] * [items.quantity]
```

How to create a computed field using Query Editor—Columns

- 1 In Query Editor, choose Columns.
Query Editor—Columns appears.
- 2 In Column Name, type a name for the computed field.

When you select another field, Query Editor inserts the following text at the beginning of the name that you typed:

<Computed>.

- 3 In Formula, type the expression that computes the value of the field, as shown in Figure 8-17. For more information about the syntax to use, see “Using QBE to specify a condition,” later in this chapter.



To use expression builder to create the expression, choose Ellipsis.

If the expression evaluates to a string, the string must not exceed 1021 characters. If the string exceeds 1021 characters, e.Report Designer Professional displays an error.

Column Name	Actuate Type	Formula
orders.issue	String	
orders.askByDate	Date	
orders.custID	Integer	
orders.category	Double	
<Computed>.Amount	Double	items.pricequote*items.quantity

Figure 8-17 Computing the value of the Amount field

If you are planning to use aggregate rows in the query, use one of the aggregate functions: Sum(), Min(), Max(), Ave(), First(), Last(), or Count(). For more information about aggregate functions, see *Developing Reports using e.Report Designer Professional*.

- 4 In Actuate Type, select a data type from the drop-down list.

You do this step last step, because changing a formula causes Actuate e.Report Designer Professional to reset the Actuate Type to Variant.

Changing the order of columns or the data type of a column

You can specify the order of columns in a query by choosing them in the desired order or by changing the order after you create the query. You can also modify the Actuate Basic data type of the column.

How to change column order or data type

- 1 In Query Editor, choose Columns.
- 2 On Query Editor—Columns, to change the order in which the columns appear, select a column name and choose the up arrow or down arrow to move the column.
- 3 To change the Actuate Basic data type of a column in the report’s data row, complete one of the following tasks:
 - Double-click the column name in the upper part of Query Editor.
In Column Property, the Database Type field displays the data type of the column in the data source.

In Actuate Basic Type, select an Actuate Basic data type, as shown in Figure 8-18.

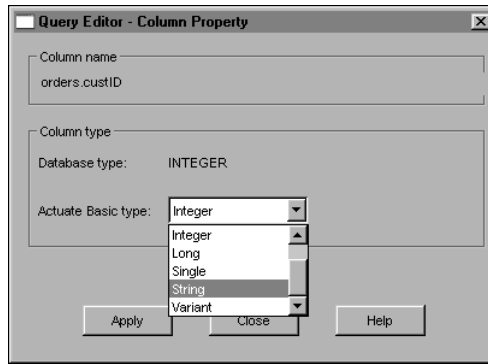


Figure 8-18 Selecting an Actuate Basic data type

- In Query Editor—Columns, click a field under Actuate Type, and select a data type from the drop-down list, as shown in Figure 8-19.

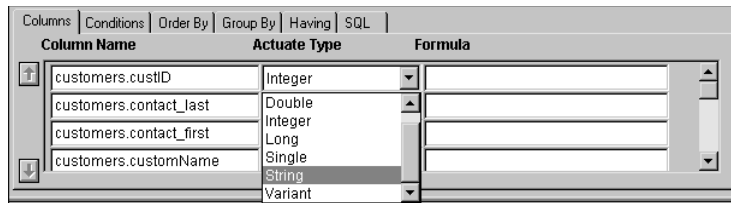


Figure 8-19 Selecting a data type

Summarizing data from multiple rows

To summarize a group of retrieved data rows, create an aggregate row. An aggregate row is a single row that combines the data from a group of rows when those rows have one or more columns in common. For example, you can group all rows for a particular customer and create one aggregate row that contains the total number and average dollar amount of that customer's orders. You also could group items by category and display the average price for items in that category. For more information about aggregate functions, see *Developing Reports using e.Report Designer Professional*.

As you write an aggregate expression on Query Editor—Columns, e.Report Designer Professional adds it to the SELECT clause of the SQL SELECT statement. As you select columns for aggregate rows on Query Editor—Group By, e.Report Designer Professional writes the GROUP BY clause of the SQL SELECT statement.

To create an aggregate row, you must complete the following tasks in this order:

- Specify the grouping column and one or more aggregate expressions on Query Editor—Columns. Query Editor—Columns must contain only the group column and aggregate expressions. The SQL language does not permit the

inclusion of any other columns unless they are from another table and do not conflict with the grouping. For more information about including additional columns, see a SQL manual or the documentation for your database or ODBC data source.

- Ensure that your data is sorted by the grouping column. Summarizing data does not sort the results. Your query must sort the records by the grouping column for summarization to return the proper results. One method of sorting is to specify sorting by the grouping column on Query Editor—Order By. For more information about the need for sorting when summarizing data, see a SQL manual or the documentation for your database or ODBC data source. For more information about sorting in e.Report Designer Professional, see “Sorting data” in Chapter 1, “Accessing a data source” and “Specifying sorting for a textual query,” later in this chapter.
- Select a grouping column on Query Editor—Group By. The query returns only one row for each different value in the grouping column.

How to summarize data

- 1 In Query Editor, choose Columns.

Query Editor—Columns appears.

- 2 Select the grouping column by dragging a column from the upper part of Query Editor and dropping it in Column Name.

You can choose additional columns if you also choose them in Query Editor—Group By and if using all of the columns together to group records does not cause a SQL error.

- 3 Set up the aggregate expression:

- 1 In Column Name, type a name for the aggregate expression.
- 2 In Formula, type the expression that computes the value of the column. Use one of the aggregate functions, such as Sum(), Min(), Max(), Ave(), First(), Last(), or Count().



If you want more room to type the expression, choose Ellipsis to open expression builder.

If the expression evaluates to a string, the string must not exceed 1021 characters. If the string exceeds 1021 characters, e.Report Designer Professional displays an error.

- 3 In Actuate Type for the aggregate expression, select a data type from the drop-down list.

Do not select a data type until you define a formula. e.Report Designer Professional sets the data type of a column to Variant when you define a formula.

- 4 Repeat steps 2–3 for any additional aggregate expressions that you want to display.
 - 5 In Query Editor, choose Group By.
If the Group By tab is hidden:
 - 1 Choose Tools→Options.
Options—Design Editor appears.
 - 2 Choose Query Editor.
Options—Query Editor appears.
 - 3 Select Enable Group By and Having editors.
Choose OK.
 - 4 In Query Editor, choose Group By.
Query Editor—Group By appears.
 - 6 In Available Columns, select the grouping column for the aggregate rows.
You can select additional columns for grouping if Query Editor—Columns also contains the columns and if using all of the columns together to group records does not cause a SQL error.
 - 7 Choose the left arrow.
- The column names appear in Group By, as shown in Figure 8-20.

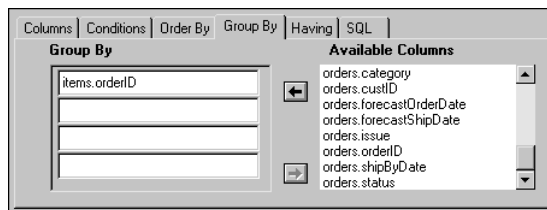


Figure 8-20 Selecting grouping columns

How to remove a grouping column for summarized data

- 1 In Query Editor, choose Group By.
Query Editor—Group By appears.
- 2 In Group By, select the column name. Press Shift+Delete.
- 3 Choose Columns.
Query Editor—Columns appears.
- 4 Select the column name. Press Shift+Delete.

- 5 If the deleted column was the only grouping column, delete any computed fields that contain aggregate expressions.

Adding conditions to a query

You can impose two kinds of conditions on a query:

- Conditions on row retrieval. To select data based on values in individual rows in the database or ODBC data source, use row retrieval conditions. Specify these conditions on Query Editor—Conditions. e.Report Designer Professional adds these conditions to the WHERE clause of the SQL SELECT statement that it generates. For more information about adding conditions on row retrieval, see “Putting a condition on row retrieval,” later in this chapter.
- Conditions on aggregate rows. To select data based on values that are not known until after the query creates aggregate rows, such as total order amounts, use aggregate row conditions. Specify these conditions on Query Editor—Having. e.Report Designer Professional adds these conditions to the HAVING clause of the SQL SELECT statement that it generates. For more information about adding conditions on an aggregate row, see “Putting conditions on an aggregate row,” later in this chapter.

There are two ways to put conditions on row retrieval. You can use either or both of them in a single query:

- Use SQL syntax to specify a condition that involves more than one column. You also can specify an OR or AND condition. For information about SQL syntax, see a SQL manual or the documentation for your database or ODBC data source.
- Use Query by Example (QBE) expression syntax. Each condition that you specify using QBE can involve only one column. When you specify more than one condition using QBE syntax, all conditions must be True for a particular row to be returned. That is, you can specify only AND conditions. For more information about specifying conditions using QBE, see “Using QBE to specify a condition,” later in this chapter.

For information about enabling users to specify the value of a column as a condition, see Chapter 9, “Filtering data.”

For information about enabling users to specify an expression as a condition, see Chapter 9, “Filtering data,” later in this chapter.

About how e.Report Designer Professional applies conditions

When a query with both row retrieval and aggregate conditions runs, e.Report Designer Professional applies the conditions in the following order:

- The query retrieves all rows that meet the row retrieval conditions, as shown in Figure 8-21.

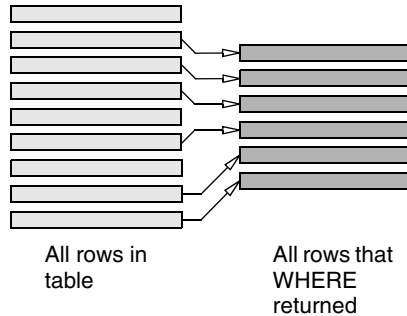


Figure 8-21 Filtering rows

For example, the query selects salary numbers for all departments within a division of the company.

- The query groups those rows by the grouping columns and creates aggregate rows that summarize the groups, as shown in Figure 8-22.

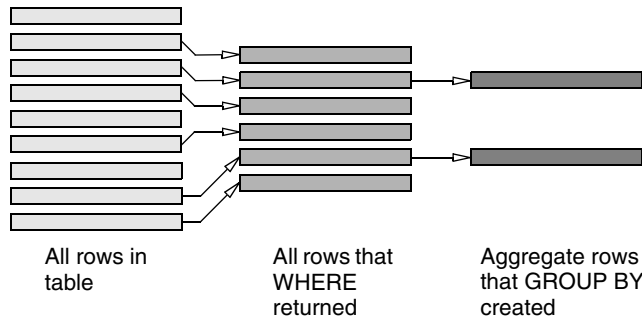


Figure 8-22 Aggregating rows

For example, the query combines the salary numbers by department and creates rows that contain the average salary for each department.

- The query applies the conditions for the aggregate rows that it created, as shown in Figure 8-23.

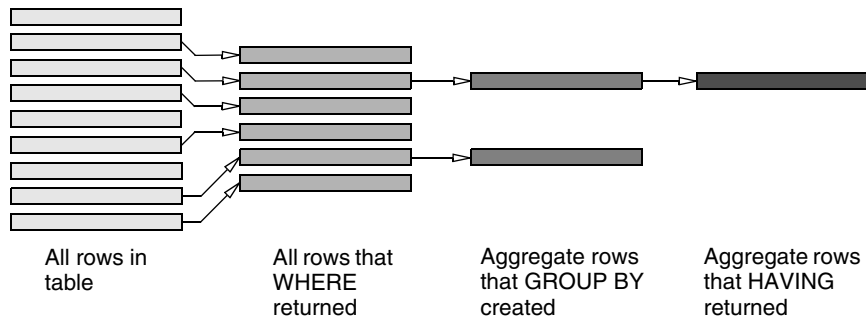


Figure 8-23 Filtering the aggregate rows

For example, the query includes only those departments for which the average salary exceeds a particular amount.

Using QBE to specify a condition

Actuate e.Report Designer Professional provides a simple Query by Example (QBE) syntax with which you can write an expression. e.Report Designer Professional translates the expression into SQL. QBE syntax applies only in a report that uses a query data stream. Use QBE syntax in the following situations:

- To set conditions on row retrieval on Query Editor—Conditions
- To set conditions on aggregate rows on Query Editor—Having
- To choose values for ad hoc parameters on the Actuate Information Console Requester page

When you use QBE syntax, e.Report Designer Professional reads the QBE expression and adds the corresponding SQL code to the query. You can build a variety of expressions using QBE syntax, including the following types:

- A single value, such as 10
- A relational expression, such as >10
- A range of values, such as 10–20
- A list of values, expressions, or ranges that are separated by pipe symbols, such as 10 | 20–30 | >50
- A group of values, such as (abc | xyz), that can be combined in a Boolean expression, such as (abc | xyz)&bbb.

For information about QBE expressions and operators, see *Using Information Console*.

Putting a condition on row retrieval

You can specify one or more conditions that a row must meet for a query to return that row. For example, rather than retrieving information about all customers, you can retrieve information about particular kinds of customers. As you put conditions on row retrieval, Actuate e.Report Designer Professional adds them to the WHERE clause of the SQL SELECT statement.

You can write a condition on row retrieval in QBE or SQL syntax. For more information about QBE syntax, see “Using QBE to specify a condition,” earlier in this chapter. A SQL expression can include any valid SQL syntax, including nested SELECT statements. For more information about SQL syntax, consult a SQL manual or the documentation for your database or ODBC data source.

You can also have users specify conditions when they run the report. For more information about having users specify conditions, see Chapter 9, “Filtering data.”

If you want users to type a single value to complete the expression when they run the report, create a SQL or QBE expression that uses a static parameter. For more information about static parameters, see Chapter 9, “Filtering data.”

How to put conditions on row retrieval using QBE

- 1 In Query Editor, choose Conditions.

Query Editor—Conditions appears.

- 2 On Query Editor—Conditions, click under Column Name, and select a column name from the drop-down list.

For example, in Column Name, choose items.pricequote.

- 3 In Query Expression, type a QBE expression for your condition. To have more space and help for creating a QBE expression, choose Ellipsis to open expression builder. Ellipsis only appears when the cursor is in Query Expression.



For example, Figure 8-24 shows a QBE expression that selects data rows from which the value of items.pricequote exceeds 100.

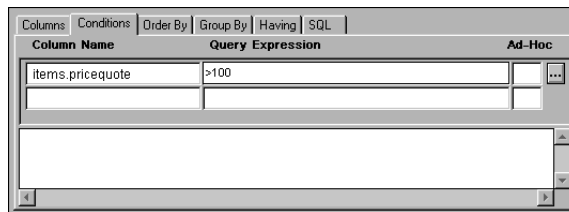


Figure 8-24 A filter condition using a QBE expression

- 4 To view the results for the WHERE clause, choose SQL.
Query Editor—SQL appears.
- 5 To add more conditions, choose Conditions, and repeat steps 2–3.

How to delete a QBE row retrieval condition

- 1 In Query Editor, choose Conditions.
Query Editor—Conditions appears.
- 2 Select the column name. Press Shift+Delete.

How to put conditions on row retrieval using SQL

- 1 In Query Editor, choose Conditions.
Query Editor—Conditions appears.
- 2 In the bottom field, type a SQL expression for the condition.

For example, specify in SQL that the value of `items.pricequote` must be greater than 100, as shown in the following code and Figure 8-25:

```
items.pricequote>100
```

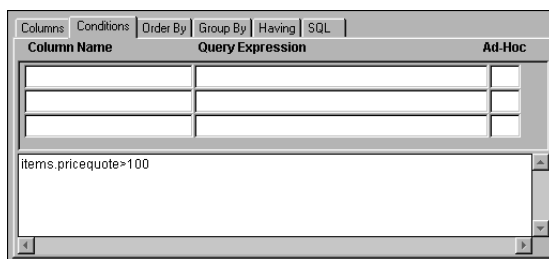


Figure 8-25 Using Actuate SQL to specify a filter condition

- 3 To view the resulting WHERE clause, choose SQL.
Query Editor—SQL appears.
- 4 To add more conditions, choose Conditions, and repeat step 2.

How to delete a SQL row retrieval condition

- 1 In Query Editor, choose Conditions.
Query Editor—Conditions appears.
- 2 Select the SQL expression. Press Shift+Delete.

Putting conditions on an aggregate row

You can set conditions that an aggregate row must meet. For example, rather than including aggregate rows for all customers, you can include only customers with order totals of more than \$10,000. As you put conditions on aggregate rows, e.Report Designer Professional writes the HAVING clause of the SQL SELECT statement.

For more information about how to specify aggregate rows, see “Summarizing data from multiple rows,” earlier in this chapter. For more information about conditions, see “About how e.Report Designer Professional applies conditions,” earlier in this chapter.

You can write a condition on aggregate rows in QBE or SQL syntax. A SQL expression can include any valid SQL syntax, including nested SELECT statements. For more information about SQL syntax, consult a SQL manual or your database or ODBC data source documentation. For more information about QBE syntax, see “Using QBE to specify a condition,” earlier in this chapter.

You can also have your users specify conditions when they run the report. For more information about having users specify conditions, see Chapter 9, “Filtering data.”

If you want users to type a single value to complete the expression when they run the report, create a SQL or QBE expression that uses a static parameter. For more information about static parameters, see Chapter 9, “Filtering data.”

How to put conditions on aggregate rows using QBE

- 1 In Query Editor, choose Having.

Query Editor—Having appears.

If Query Editor—Having is hidden, see “How to summarize data,” earlier in this chapter, for more information about displaying Group By and Having.

- 2 Click under Column or Aggregate, and select an aggregate column name from the drop-down list.

The drop-down list displays the grouping columns that are specified on Query Editor—Grouping and the aggregate columns that are specified on Query Editor—Columns. The aggregate column names are prefaced by <Computed>. to indicate that they are computed fields. For example, in Column or Aggregate, choose <Computed>.OrderTotal.



- 3 In Query Expression, type a QBE expression for your condition. To have more space and help for creating a QBE expression, choose Ellipsis to open expression builder.

For example, you can specify in QBE that the value of OrderTotal must be greater than 1000000, as shown in the following code and Figure 8-26:

```
>1000000
```

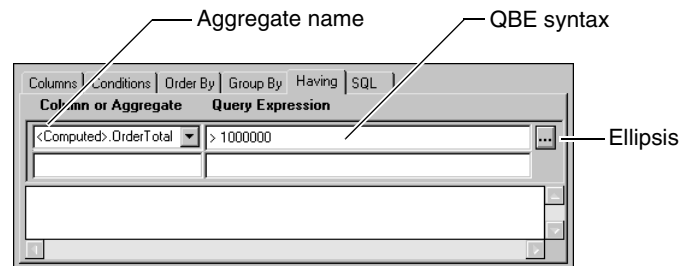


Figure 8-26 An aggregate filter condition using a QBE expression

- 4 To view the resulting HAVING clause, choose SQL:

```
HAVING (items.pricequote * items.quantity) > '1000000'
```

- 5 To add more conditions, choose Having, and repeat steps 2–3.

How to delete a QBE aggregate row condition

- 1 In Query Editor, choose Having.

Query Editor—Having appears.

If Query Editor—Having is hidden, see “How to summarize data,” earlier in this chapter, for more information about displaying Group By and Having.

- 2 Select the column name. Press Shift+Delete.

How to put conditions on aggregate rows using SQL

- 1 In Query Editor, choose Having.

Query Editor—Having appears.

If Query Editor—Having is hidden, see “How to summarize data,” earlier in this chapter, for more information about displaying Group By and Having.

- 2 In the bottom field, at the bottom of Query Editor—Having, type a SQL expression for the condition.

For example, specify in SQL that the value of the sum of all price quotes, multiplied by the number of items in the order, must be greater than 1000000, as shown in the following code and Figure 8-27:

```
sum(pricequote * itemcount)>1000000
```

- 3 To view the resulting WHERE clause, choose SQL.

Query Editor—SQL appears.

- 4 To add more conditions, choose Having, and repeat step 2.

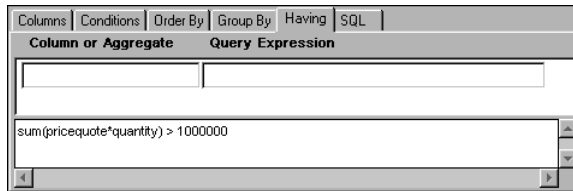


Figure 8-27 An aggregate filter condition using Actuate SQL

How to delete an aggregated row condition written in SQL

- 1 In Query Editor, choose Having.

If Query Editor—Having is hidden, see “How to summarize data,” earlier in this chapter, for more information about displaying Group By and Having.

- 2 Select the SQL expression. Press Shift+Delete.

Viewing the generated SQL SELECT statement

A SELECT statement specifies which data to retrieve from the data source.

A SQL SELECT statement consists of parts called clauses. Table 8-2 shows the relationship between actions in Query Editor and clauses in the generated SQL SELECT statement.

Table 8-2 Relationship between Query Editor actions and SQL SELECT statement clauses

Action in Query Editor	SQL SELECT statement clause
Using tables, views, and synonyms	FROM and WHERE
Selecting and modifying columns and creating computed fields	SELECT
Putting a condition on row retrieval	WHERE
Specifying sorting for a textual query	ORDER BY
Summarizing data from multiple rows	GROUP BY
Putting conditions on an aggregate row	HAVING
Prompting the user to provide a specific value to filter results	FROM, WHERE, or HAVING
Filtering query results with user-supplied expressions	FROM, WHERE, or HAVING

For more information about how to perform these actions in Query Editor, see the corresponding section in this chapter or in Chapter 9, “Filtering data.”

How to view the SELECT statement

- 1 In Query Editor, choose SQL.
Query Editor—SQL appears.
- 2 Review the SQL SELECT statement that Actuate e.Report Designer Professional wrote in response to your choices in Query Editor. An example is shown in Figure 8-28.

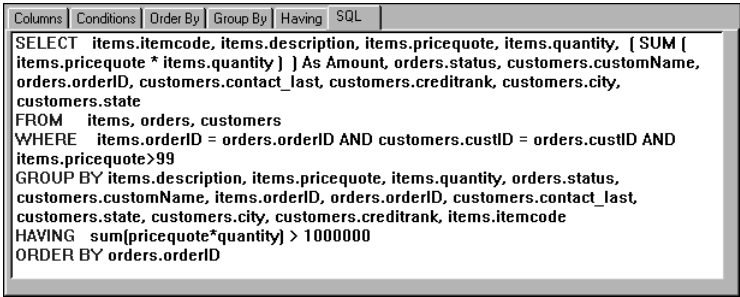


Figure 8-28 The resulting SQL SELECT statement

How to copy the SELECT statement to Clipboard

You can copy the SQL SELECT statement to Clipboard to edit it manually or to paste it elsewhere.

On Query Editor—SQL, select the text you want to copy. Choose Edit→Copy.

Creating a query by writing a SQL SELECT statement

To create a query by writing SQL code, you perform the following procedures in this order:

- Writing the SQL query
- Preparing the query
- Modifying the textual query, if desired
- Accepting the textual query

The following sections describe each of these procedures.

Writing the SQL query

To choose the data to include in the report, you write a query. A query is a SQL SELECT statement that specifies what data to retrieve from a data source.

The Textual Query Editor supports tab and new-line characters in the query string. This ability supports pasting query text directly from an external application, such as a database query tool. You can also include comments in your SQL query in the format that your database supports.

If you plan to use stored procedures that are provided by your database, use Stored Procedure Data Source Builder. For more information about working with stored procedures, see Chapter 11, “Accessing data using a stored procedure.”

For information about using parameters in a query, see Chapter 9, “Filtering data.”

How to write a query using Textual Query Editor

- 1 Create a textual query data source.

To write the SQL code for your query, you must have a textual query data source. For information about creating a query as a textual query data source, see “Creating a query data source” in Chapter 7, “Connecting to a database or ODBC data source.” For information about converting a graphical query data source into a textual query data source, see “Converting a graphical query to a textual query,” later in this chapter.

- 2 Choose View→Data to open Textual Query Editor.

Textual Query Editor appears, as shown in Figure 8-29.

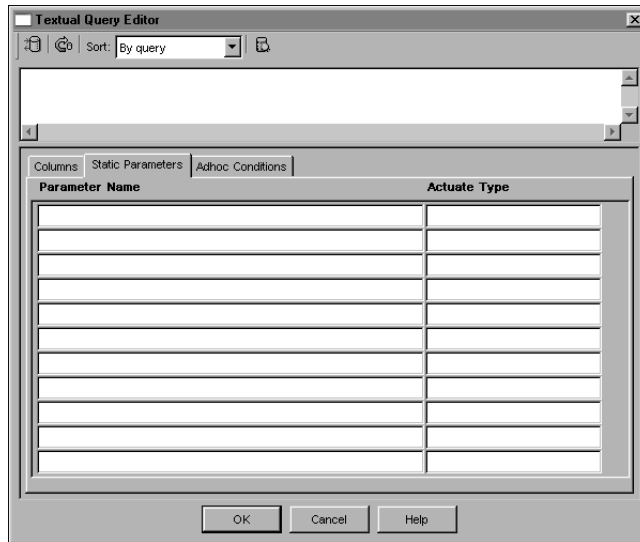


Figure 8-29 Textual Query Editor

- 3 In the upper part of Textual Query Editor, type the SQL SELECT statement.

If your data source login requires it, fully qualify the table name with the name of the data source. For example, to select all columns from the customers table of the Actest database, you can use the fully qualified name, Actest.customers, in the SELECT statement. You do not have to specify the data source name in subsequent clauses of the query, such as the WHERE clause.

To retrieve data for only those customers in Massachusetts, specify State in the WHERE clause of the SELECT statement, as shown in Figure 8-30.

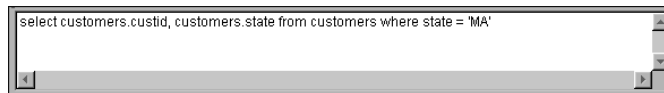


Figure 8-30 A SQL SELECT statement

While editing your SQL code, you can use standard text editing functionality, such as copy and paste. You can access these commands and others on the context menu. Right-click in the upper part of Textual Query Editor to display the context menu, as shown in Figure 8-31.

Choose Undo to undo your editing of the SQL SELECT statement. Undo affects only the query that you type. It does not apply to any other Textual Query Editor function, such as Describe Query or Clear Query.

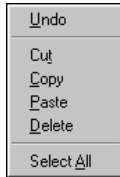


Figure 8-31 Textual Query Editor's context menu



You also can choose Clear Query to erase the entire SQL code. This also clears the lower half of Textual Query Editor, erasing any information about the query that you specified there.



If you want to preview the results of your query, choose Preview Data. For more information about previewing data in a database query, see “Previewing the rows that a database or ODBC query returns,” later in this chapter.

Preparing the query



Choose Describe Query to prepare the SQL SELECT statement. Describe Query sends the SQL SELECT statement to the connected data source server to describe the query's result columns. If you use the select all character, which is an asterisk (*), in your SQL SELECT query, e.Report Designer Professional retrieves information for all columns in the table. It also updates the relevant information in the lower part of Textual Query Editor.

Activate e.Report Designer Professional sends the SQL SELECT statement to your connected data source server. All data source connections return the result columns information to Textual Query Editor.

If e.Report Designer Professional cannot connect to the data source, Database Login appears, as shown in Figure 8-32.

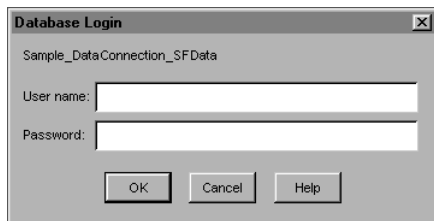


Figure 8-32 Database Login

If Database Login appears, type the required information, and choose OK. If you provided connection information in the connection component properties for the report, e.Report Designer Professional provides that information for you. For more information about defining connection properties, see “Defining a database or ODBC connection” in Chapter 7, “Connecting to a database or ODBC data source.”

Figure 8-33 shows Textual Query Editor with the result column information.

Columns Static Parameters Adhoc Conditions				
Column Name	DB Type	Actuate Type	Display Length	Reference Names
custid	INTEGER	Integer	11	custid
state	VARCHAR	String	2	state

Figure 8-33 Result columns

Modifying the textual query

Check the properties in the lower part of Textual Query Editor, and modify them as necessary. You can modify this textual query in the following ways:

- Changing the properties of a result column
- Specifying sorting for a textual query
- Prompting the user to provide a specific value to filter results
- Filtering query results with user-supplied expressions

For more information about these optional steps, see the corresponding section, later in this chapter, or in Chapter 9, “Filtering data.”

Accepting the textual query

- 1 Check the properties in the lower part of Textual Query Editor and verify that they are appropriate.

For information about understanding and modifying Textual Query—Columns, see “Changing the properties of a result column,” later in this chapter. For information about understanding and modifying Textual Query—Static Parameters, see Chapter 9, “Filtering data.” For information about understanding and modifying Textual Query—Adhoc Conditions, see “Filtering query results with user-supplied expressions” in Chapter 8, “Creating a database or ODBC query.”

- 2 Choose OK to accept the query and its column and parameter properties.

Converting a graphical query to a textual query

To modify a SQL SELECT statement that you created graphically in Query Editor, you must convert the graphical query to a textual query. You cannot undo this conversion.

If you use the FetchLimit property for a data source component, you can adjust its value after converting a graphical query to a textual query. FetchLimit controls

the number of rows that e.Report Designer Professional returns from the data source. If you add a filtering condition, e.Report Designer Professional adds the condition to the graphical query before the SQL statement is sent to the data source. As a result, a graphical query only returns rows that passed the filtering condition. A textual query is sent to the database exactly as specified, and the filtering conditions are applied after fetching the rows. In the end, both types of queries fetch the same number of rows from the data source, but some rows that the textual query fetches may be eliminated by the filtering condition.

How to convert a graphical query to a textual query

- 1 In e.Report Designer Professional, choose View→Data.
- 2 In Query Editor, choose SQL→Edit SQL.

A warning, shown in Figure 8-34, reminds you that you cannot convert the query back to a graphical format.

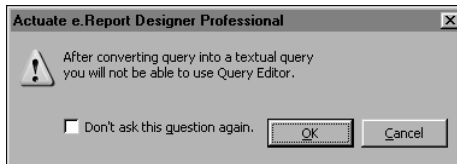


Figure 8-34 Warning about converting a graphical query to a textual query

- 3 Choose OK.

Textual Query Editor opens and displays the query, as shown in Figure 8-35.

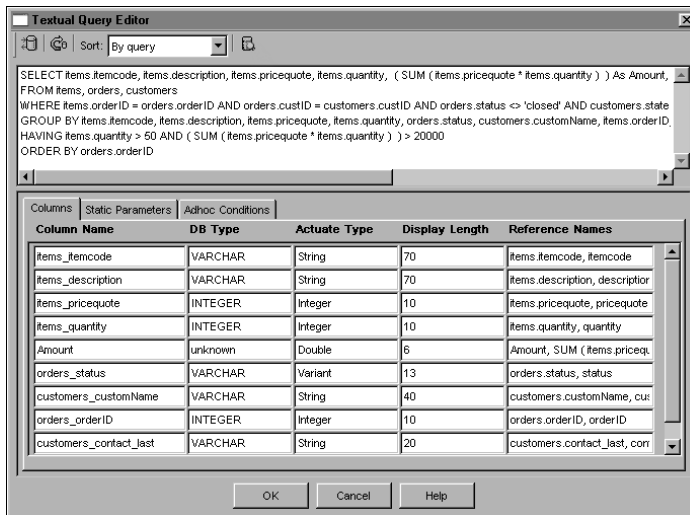


Figure 8-35 Textual Query Editor, displaying a converted graphical query

- 4 If two tables have the same column name, modify the default reference names as necessary. For more information about changing reference names, see “Changing the properties of a result column,” later in this chapter.
- 5 In Reference Names, add a reference name to the beginning of the list using the following format:

`tablename.columnname`

This change ensures that e.Report Designer Professional correctly identifies the table for each column when choose Describe Query to prepare the query. During conversion, this information is embedded in the column name in the following format:

`tablename_columnname`

For example, the customName column in the customers table appears as follows:

`customers_customName`

When you later choose Describe Query, the design tool queries the database or ODBC data source for the column names and replaces them on Textual Query Editor—Columns. The database or ODBC data source returns only the column names, not the table name, as metadata when e.Report Designer Professional describes the query. For example, the customers_customName entry in Column Name becomes customName. If you include a reference name using the tablename.columnname format, your query continues to associate the columns with their tables.

After you convert a query to a textual query, you can modify it as described in “Creating a query by writing a SQL SELECT statement,” earlier in this chapter.

Previewing the rows that a database or ODBC query returns

You can preview the results of a graphical or textual query without building and executing the report. Previewing the query enables you to determine whether you want to add any additional conditions to the query or modify the query in other ways.

You also can preview the data in a single table or column as you design the query. For information about previewing the data in a table, see “Creating a query by writing a SQL SELECT statement,” earlier in this chapter. For information about previewing the data in a table, see “How to specify that the query only returns distinct rows,” earlier in this chapter.

How to preview a database or ODBC query

- 1 If you have a graphical query, choose SQL→Preview Data.



If you have a textual query, choose Preview Data.

If the query uses any parameters, Specify Parameter Values appears.

If the query does not use parameters, Preview Query Data appears.

- 2 If the query uses parameters, specify the desired values for the parameters on Specify Parameter Values, as shown in Figure 8-36. When you are finished, choose OK.

Name	Type	Value
customers_creditrank	String (ad hoc)	
customers_customName	String (ad hoc)	
customers_purchaseFrequency	String (ad hoc)	
customers_purchaseVolume	String (ad hoc)	
offices_city	String (ad hoc)	
offices_state	String (ad hoc)	
orders_forecastOrderDate	Date (ad hoc)	
orders_forecastShipDate	Date (ad hoc)	
orders_orderID	Integer (ad hoc)	
orders_status	String (ad hoc)	
salesreps_first	String (ad hoc)	
salesreps_last	String (ad hoc)	

Figure 8-36 Specifying the parameter values

An asterisk (*) indicates a required parameter. If the parameter has a default value, e.Report Designer Professional displays the default value for the parameter when you preview the query data for the first time. After the first time, e.Report Designer Professional displays the most recent parameter values for the query.

If you enter an invalid value for the parameter, e.Report Designer Professional tries to correct it. For example, if the parameter is for an integer field, e.Report Designer Professional corrects “123abc” to “123”. If e.Report Designer Professional cannot correct the value, it changes it to a blank value.

If a parameter value is blank, e.Report Designer Professional substitutes the default value, if one exists. e.Report Designer Professional runs the query.

Preview Query Data appears.

- 3 On Preview Query Data, shown in Figure 8-37, scroll horizontally and vertically to see the data returned from the query.

The status bar at the bottom left lists performance statistics. For long-running queries, you can use these statistics to help tune your query. For more information about tuning a query, see “Timing and optimizing data source queries” in Chapter 10, “Maintaining a database or ODBC query.”

	itemcode	Expr1001	description	pricequote	quantity
1	MP1632	18731 Douglas Av.	32 bit Programmabl...	210	49
2	MR3240	18731 Douglas Av.	32M x 4 Dynamic R...	31	50
3	MRL0480	18731 Douglas Av.	4M x 8 Dynamic Ra...	18	49
4	MRL3240	18731 Douglas Av.	32M x 4 Dynamic R...	61	49
5	MS0880	18731 Douglas Av.	8M x 8 Static Ram	52	49
6	MSL1680	18731 Douglas Av.	16M x 8 Static Ram...	165	50
7	MP1608	18731 Douglas Av.	8 bit Programmable,...	43	50
8	MP2032	18731 Douglas Av.	32 bit General Purp...	310	49
9	MR0890	18731 Douglas Av.	8M x 9 Dynamic Ram	33	49
10	MS3240	18731 Douglas Av.	32M x 4 Static Ram	77	49
11	MSL0890	18731 Douglas Av.	8M x 9 Static Ram, ...	165	49
12	MS0840	18731 Douglas Av.	8M x 4 Static Ram	37	50
13	MS0410	18731 Douglas Av.	4M x 1 Static Ram	10	49
14	MP1604	16780 Pompton St.	4 bit Programmable,...	21	52
15	MP1608x	16780 Pompton St.	8 bit Programmable ...	48	52
16	MP1632s	16780 Pompton St.	32 bit Programmabl...	290	52
17	MPL1632	16780 Pompton St.	32 bit Programmabl...	303	51

Figure 8-37 Previewing the query data

- Expand the width of any truncated columns by clicking the right border of the column label.

For more information about viewing the data, see “How to preview the columns and data in a table,” earlier in this chapter.

Changing the properties of a result column

You can change the Actuate data type for query results using the graphical or textual query editor. You can also change the default display length or the reference name of query results using the textual query editor. You can change the display name for graphical and textual queries using the column editor.

Changing the data type of graphical query results

You can change how a data source type maps to an Actuate Basic data type for a graphical query. To change the data type of a column that you select for a query, change the value of Actuate Type for the column name on Query Editor—Columns.

If you select the same column more than once, the Actuate data type of that column changes for every occurrence. For example, Figure 8-38 shows the selection of a different data type for one of several identical column names.

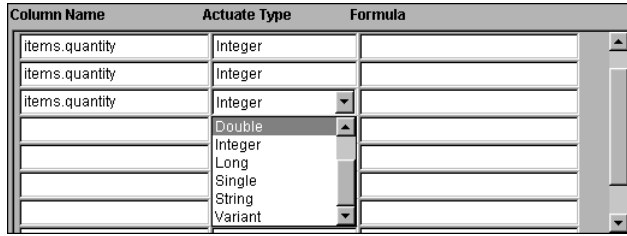


Figure 8-38 Selecting a data type for one of several identical column names

After you select a new Actuate data type, all of the identical column names have that data type, as shown in Figure 8-39.

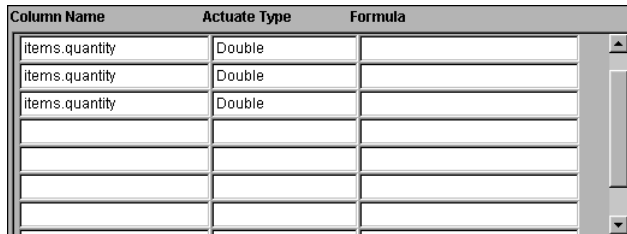


Figure 8-39 The identical column names all have the new Actuate data type

Changing the display name of query results

To change the display name of a column in query results, use Data Row Editor.

How to change the display name of query results

- 1 Right-click the data row in Report Structure, and choose Data Row Editor.
Data Row Editor appears.
- 2 In Variable Name, select a column, as shown in Figure 8-40.

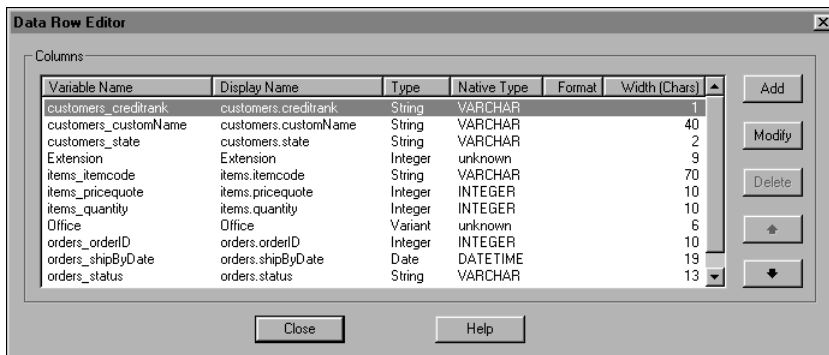


Figure 8-40 Selecting a column

Choose Modify.

Column Editor appears.

- 3 In Display name, type the desired display name, as shown in Figure 8-41.

Column Editor

Variable name: customers_creditrank

Display name: customers.ranking

Table.col: customers.creditrank

Data type: String Native data type: unknown

Format: Column can be used as custom filter

Width: 20 (number of characters) Word wrap

Text format: Plain RTF HTML

e.Analysis option: Automatic Dimension Measure

Expression:

OK Cancel Help

Figure 8-41 Typing the display name

Choose OK.

Changing the data type, display length, and reference name of textual query results

For a textual query, you can modify the Actuate data type, display length, and reference name of the columns in rows that the textual query returns.

When you prepare the textual query to use your modified SQL SELECT statement using Describe Query, it obtains and calculates information about the result columns. For example, the textual query editor obtains a result column's data type in the database and calculates a corresponding Actuate type. You find this information in Textual Query Editor on Textual Query—Columns, shown in Figure 8-42.

Columns				
Static Parameters				
Adhoc Conditions				
Column Name	DB Type	Actuate Type	Display Length	Reference Names
custid	INTEGER	Integer	11	custid
state	VARCHAR	String	2	state

Figure 8-42 Result column information

On Textual Query—Columns, you can view, but not modify, the values for the following fields:

- Column name

Typically, Describe Query reports the name of the column. If Describe Query does not report a column name, e.Report Designer Professional assigns a name in the following format:

```
column<selectOrderNumber>.
```

For example, if the column is the second column that was selected, the assigned name is column2. If column names are duplicates, e.Report Designer Professional adds a unique number to the column name. For example, Address, Address_1, and Address_2.

- DB type

Describe Query reports the column's native data source data type. For example, the data type can be Number, Varchar, or Char.

You can modify several properties of a column:

- Actuate Basic data type

After retrieving the data source type for a column, e.Report Designer Professional maps the column's data source data type to an Actuate type, such as Double, String, or Integer. Use the Actuate type drop-down list to map the column to another Actuate type.

- Display length for the column

e.Report Designer Professional displays the default display length, if any. e.Report Designer Professional uses the display length to determine the default size of the corresponding data control. You can modify the default display length by typing the desired number in the Display length field.

- Reference name for the column, such as in the field list

For a new query, the default reference name is the same as the column name. You can specify several names by separating them by commas. e.Report Designer Professional uses the first name you provide in Reference Names as the display name of a data row variable elsewhere, for example in Fields. You can use the additional reference names in methods in a report.

A typical reference name change is to include the table name in the reference name, especially when two tables include the same column name. A typical use of a second reference name is to provide a shorter name for easy use in any code that you program for the report. For example, if the default reference name is ID, you could change it to employee.ID, empid. Your users see the table name with the column name and also the shorter name, empid, in Fields. You can use either name in your methods for the report.

Specifying sorting

You need to sort data in your query or for your query if the data is not already sorted in the data source and one of the following conditions is true:

- You do not use group sections.
- You use group sections, but you want to do additional sorting.
- You wish to override the automatic sorting that the group sections provide.

For more information about sorting provided by group keys, see “Sorting data” in Chapter 3, “Modifying data processing.”

You can sort data in your query or for your query in several ways, which are described in the following list in order of precedence:

- **Group section’s sort key**
The dominant sort criteria is provided by a group section’s sort key, if one is specified. Group sections can contain a sort key that e.Report Designer Professional uses to create an ORDER BY clause for queries. For more information about overriding the automatic sorting that the group sections provide, see “Overriding sorting by the group section sort key,” later in this chapter.
- **Order By property**
You can provide additional sorting instructions using the OrderBy property. This approach does not require that you modify the query. You can use the Order by property for read-only queries from a library and queries that you specify in methods instead of in Query Editor or Textual Query Editor. For more information about sorting with an OrderBy property, see “Specifying the sort order using the OrderBy property” in Chapter 3, “Modifying data processing.”
- **ORDER BY clause in the query**
You can change the query to specify that the data source sorts the data before it sends the data to e.Report Designer Professional. The procedures for specifying sorting in a query depend on whether the query is a graphical query or a textual query. For more information about sorting within a graphical query, see “Specifying data source sorting using a graphical query,” later in this chapter. For more information about sorting within a graphical query, see “Specifying sorting for a textual query,” later in this chapter.

A database can sort data so do not create a custom sort filter in e.Report Designer Professional for a database. A custom sort filter is more appropriate for sorting data from other types of sources, such as flat files. For more information about general sorting issues, see “Sorting data,” in Chapter 3, “Modifying data processing.”

Specifying data source sorting using a graphical query

You can change the order of the rows a query retrieves by specifying sorting for one or more columns in a graphical query. As you select columns for sorting using Query Editor, e.Report Designer Professional adds them to the ORDER BY clause of the SQL SELECT statement.

If your report includes group sections, the data source sorts by the group section sort key, then it sorts by additional columns that you specify in the graphical query. For information about overriding group section sorting, see “Overriding sorting by the group section sort key,” later in this chapter.

For an overview of how to handle sorting with data source queries, see “Specifying sorting,” earlier in this chapter. For more information about group sections, see *Developing Reports using e.Report Designer Professional*.

How to specify data sorting using Query Editor

- 1 In Query Editor, choose Order By.
Query Editor—Order By appears.
- 2 In Available Columns, select one or more columns as a sort key.
- 3 Choose the left arrow.
The column names appear in Order By.
- 4 To change the order of the columns in the sort key, in Order By, select the column name. Choose the up arrow or down arrow.
- 5 To sort a column in descending order, deselect Asc for that column, as shown in Figure 8-43.

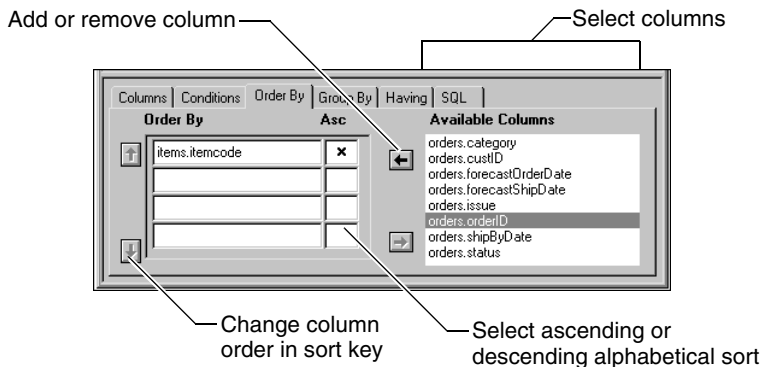


Figure 8-43 Specifying data sorting

How to stop sorting by a particular column

To stop sorting by a column, select the column on Query Editor—Order By, and press Shift+Delete.

Specifying sorting for a textual query

If your report includes group sections, e.Report Designer Professional sorts data primarily by the group section sort key, then it sorts by additional columns that you specify in the ORDER BY clause of your textual query. For information about overriding the automatic sorting that group sections provide, see “Overriding sorting by the group section sort key,” later in this chapter.

In Textual Query Editor, specify whether you want e.Report Designer Professional to:

- Determine the necessary sorting, and add an appropriate ORDER BY clause to the query. Use the By Query option when you want e.Report Designer Professional to add a dynamic ORDER BY clause to the query at run time. This is the default sort option setting. For more information about editing SQL in Textual Query Editor, see “Changing the properties of a result column,” earlier in this chapter.
- Assume that the data is sorted before it reaches e.Report Designer Professional. Use the Presorted option if your SQL query specifies sorting columns and order in an ORDER BY clause.
- Rely on a sort filter. Use the By Filter option to use e.Report Designer Professional’s sort filter utility to sort the results data. Specify the sort option that you need by selecting an option from the Sort drop-down list in Textual Query Editor, as shown in Figure 8-44.

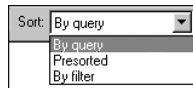


Figure 8-44 Selecting a sort option

For more information about sorting data in a report design, see “Sorting data” in Chapter 8, “Creating a database or ODBC query.”

Overriding sorting by the group section sort key

Group sections contain a sort key. e.Report Designer Professional adds this sort key to the ORDER BY clause for a query. Typically, you do not override this functionality. If you do not want e.Report Designer Professional to add the sort key to the ORDER BY clause, you can specify that the data is presorted. Use this functionality when:

- The query from a read-only library requires no modification.

- The query performs specialized sorting, and you do not want e.Report Designer Professional to modify the sorting behavior that is specified in the query.

You can also modify the default behavior of how group section sort keys modify the query that is sent to the data source. For more information about customizing the effect of group section sort keys, see “Sorting data” in Chapter 3, “Modifying data processing.”

How to use presorted data



- 1 Right-click the report section component in Report Structure, and choose Properties.

The Properties page for the component appears.

- 2 For the Sorting property, choose Presorted from the drop-down list shown in Figure 8-45.



Figure 8-45 Indicating that the data is sorted in the data source

- 3 If you use a graphical query, go to Query Editor—Order By to provide the required row order. For more information about specifying sorting for a graphical query, see “Specifying data source sorting using a graphical query,” earlier in this chapter.

If you use a textual query, type the appropriate ORDER BY clause in Textual Query Editor. For more information about editing SQL in Textual Query Editor, see “Specifying sorting for a textual query,” earlier in this chapter.

Filtering data

This chapter contains the following topics:

- About user-specified filtering in database and ODBC queries
- Prompting the user to provide a specific value to filter results
- Filtering query results with user-supplied expressions

About user-specified filtering in database and ODBC queries

When you build a query, you make the first pass at extracting certain data from the data source to use in your report. A report user can filter the data further if you create parameters that prompt the user to specify what to include in the report. You create one parameter for each question you want the user to answer. In a sales report, for example, you might create the following parameters: region, sales representative, customer name, customer credit rank, and customer purchase volume.

For each parameter, you can specify that the user needs to type the desired choice or select from a list of choices. If you want to provide a list, you can type the list of values or specify a query for e.Report Designer Professional to get the current list of values from the data in an information object column. With either type of list, you can enable the user to choose a parameter value directly or by choosing a corresponding display name for that parameter value.

The more parameters you create, the more control you give each user to generate specialized reports. Theoretically, you can create a parameter for each discrete piece of data but, for parameters to be effective and not overwhelming to the user, limit parameters to important fields. If you give the user control of what data appears in the report, consider also creating a parameter to prompt the user for a report title that describes the report's contents. You can create a parameter to prompt for a report title. This section describes how to use parameters in queries. For general information about creating and using parameters, see *Developing Reports using e.Report Designer Professional*.

For databases and information objects, you can create two types of parameters for filtering data:

- A parameter that gives the user the option to specify an expression. For example, you can create a parameter that asks the user to enter a customer state and enable the user to enter an expression, such as CA, !CA, CA | WA, or >M.
- A parameter that prompts the user to provide a specific value. Using the customer state example, such a parameter requires the user to provide a specific value that exactly matches a state name in the data source, such as CA, OR, or WA.

Use parameters in a query to support users specifying filter conditions for a query. You can create parameters to support users specifying values or expressions to filter the results of the query.

Prompting the user to provide a specific value to filter results

Use parameters that prompt the user to provide a specific data value, instead of an expression, when it makes sense to limit the parameter to a single value. Examples of such requirements are minimum quantity and maximum quantity parameters.

This type of parameter is also suitable for answering questions such as the following:

- For which single state do you want information?
- Which single invoice do you want to examine?
- About which single customer do you want information?

You can use a static parameter to support users specifying the value of a column when they run reports. You can also have users specify values that are used in other parts of the SQL SELECT statement.

When a user runs a report that contains static parameters, the viewing tool's Requester page prompts the user for the value of each static parameter. The value is inserted into the SQL SELECT statement in place of the static parameter. You also can specify properties of the parameter, such as a default value, possible values, or that the parameter is required.

If you use the name of a parameter that does not yet exist, e.Report Designer Professional defines a local parameter. If you later define a parameter using the same name as the local parameter that e.Report Designer Professional defines, e.Report Designer Professional removes the local parameter. Do not precede the parameter name with an underscore (_).

The parameter's data type must be compatible with the data source column's data type. A static parameter can be:

- A local parameter
- An inherited parameter
- A global parameter

To use a static parameter, you perform the following steps in this order:

- Use a static parameter to specify a condition or other part of SQL. You can use a simple call or SQL syntax to specify a condition that uses a static parameter.
- If you use a textual query, prepare the textual query.
- Modify the parameter properties, as desired.

If you use an ODBC driver and want to use parameters in the SELECT statement, SQL-92 imposes limitations on where you can place a parameter in the SELECT statement. Consult the appropriate SQL-92 documentation for more information.

Using a simple call to a static parameter to specify a condition

To create a parameter that requires the user to provide a specific data value, you use Query Editor—Conditions to enter an expression that defines what value the user must enter at run time. The following is an example of a simple expression that specifies selecting all records where the value is a specific state:

```
:State
```

The colon (:) indicates that State is a parameter for which the user provides a value, not a literal value for the query. e.Report Designer Professional creates a parameter called State.

The following is an example of a more complex expression that specifies selecting all records where the value is between quantity *x* and quantity *y*:

```
:QuantityMin - :QuantityMax
```

In this case, e.Report Designer Professional creates two parameters, QuantityMin and QuantityMax.

By default, when a user runs the report, the user sees the parameter names, such as State, QuantityMin and QuantityMax. You can change the parameter display names to make them more descriptive. You can also specify default values for parameters. If you do not specify a default value for a parameter that filters data in a column and the user does not enter any value for the parameter, the report is empty.

Figure 9-1 shows the definition of the State, QuantityMin, and QuantityMax parameters in the graphical query editor.

Column Name	Query Expression	Ad-Hoc
customers.state	:State	☐
items.quantity	:QuantityMin-:QuantityMax	☐
		☐

Figure 9-1 Definition of two parameters

When a user runs the report, the Requester page provides a field in which the user can specify the state and the range of quantities to include in the report, as shown in Figure 9-2.

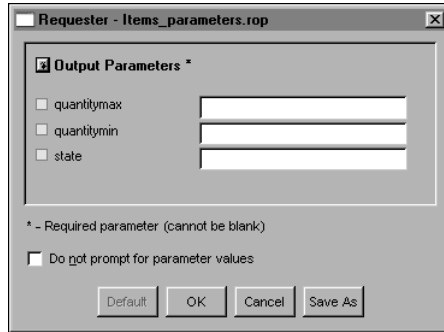


Figure 9-2 Requester page

Using SQL to specify a condition or other query clause

Whether you use a graphical query or a textual query, you can use SQL to specify a condition.

You can include a condition in a textual query by adding a WHERE, HAVING, or other clause to a SQL SELECT statement. For more information about creating a SQL SELECT statement, see a SQL manual or the documentation for your database or ODBC data source.

You can use static parameters in your condition using the single colon syntax, as shown in the following example:

```
column <comparison op> :static_param
```

where

- column is the name of the column.
- <comparison op> is a comparison operator, such as one of the following symbols:
 - =
 - >
 - <
- colon (:) specifies the location of a static parameter.
- static_param is the name of the static parameter.

For example, the following SQL SELECT statement specifies the use of a static parameter called MyCity to populate the column:

```
select city from actest.offices WHERE city = :MyCity
```

A graphical query uses a static parameter that you specify in an expression at the bottom of Query Editor—Conditions or Query Editor—Having. Figure 9-3 shows a graphical query's SQL condition that specifies the use of a static parameter called MyCity to populate the city column.

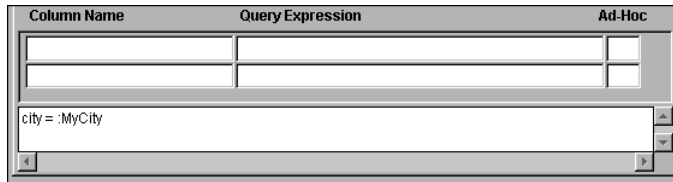


Figure 9-3 A SQL condition using a static parameter

Preparing a textual query to describe a static parameter

If you use a textual query, you must choose Describe Query after you type a condition with a static parameter. When you choose Describe Query to prepare and describe the query, Actuate e.Report Designer Professional provides information about the static parameter on Textual Query Editor—Static Parameters.

Specifying a parameter's property values

You can set the data type for a parameter in Textual Query Editor. You also can modify textual and graphical query parameters in the same way as any other parameter, by choosing Tools→Parameters, selecting the parameter, and choosing Modify. The types of properties available varies by the data type and other choices that you make for the parameter on Parameter Properties. Some of the parameter properties you can set include:

- Data type
- Display name
- Display length
- Display style
- Whether a user must supply a value
- Whether the parameter is hidden from users

This section includes instructions for changing the data type of a static parameter in Textual Query Editor. For information about using Parameter Properties to set any property values for any type of parameter, see *Developing Reports using e.Report Designer Professional*.

How to change the Actuate type for a static parameter in Textual Query Editor

To see the static parameter properties in the Textual Query Editor, choose Static Parameters. Figure 9-4 shows Textual Query Editor—Static Parameters after describing a query that uses a static parameter. The parameter name field contains the name that you used in your SQL code. Parameter names are not case-sensitive.

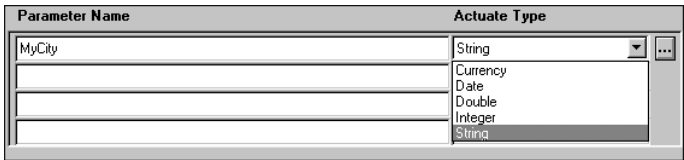


Figure 9-4 Selecting the Actuate data type for a parameter

The default Actuate type for a static parameter is String. To change the Actuate type, use the drop-down list. The parameter’s data type must be compatible with the data source column’s data type.

Filtering query results with user-supplied expressions

Create a parameter that enables the user to specify a QBE (query by example) expression if you want to provide the flexibility to enter a value, a range of values, or other filtering conditions. This type of parameter is called an ad hoc parameter. Table 9-1 shows some examples of expressions a user can enter for a customer last name parameter, and the expected results.

Table 9-1 Examples of last name parameter expressions

Expression	Gets these records
Atkins	Customers with the last name, Atkins
A-M	Customers with last names between A and M, excluding M.
Adams-Nelson	Customers with last names between Adams and Nelson
>P	Customers with last names starting with Q through Z
Peterman Firelli	Customers with the last name Peterman or Firelli

The flexibility available with this type of parameter is both an advantage and a disadvantage. The user needs to know how to construct valid expressions, and while Requester includes an expression builder to help the user build expressions, a novice report user can have trouble.

e.Report Designer Professional does not verify values that a user enters. If the user specifies an invalid expression or value, the report runs but it does not display any data. You should consider writing code to verify the values that users specify.

When a user runs a report that contains an ad hoc parameter, the viewing tool's Requester page prompts the user for input. When the user types a value for the ad hoc parameter, the Requester page adds the ad hoc parameter's corresponding conditional expression to the query's WHERE clause.

Users can specify additional conditions using Query by Example (QBE) expressions. For more information about Query by Example expressions, see *Using Information Console*.

Filtering a graphical query using a user-supplied expression

When you use an ad hoc parameter to allow or require a user to specify conditions that returned records must meet, you identify the name of a column, and the user supplies the condition for that column.

How to support using an ad hoc expression to filter a graphical query

- 1 In Query Editor, chose Conditions.
- 2 On Query Editor—Conditions, in Column Name, type or select a column name, as shown in Figure 9-5.

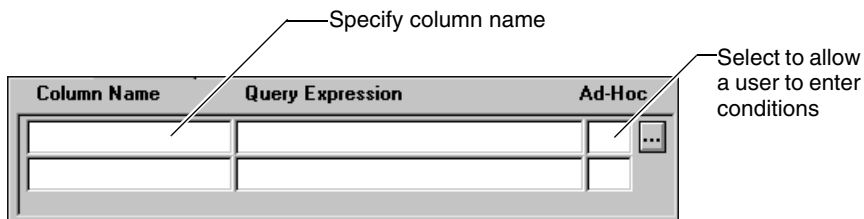


Figure 9-5 Using an ad hoc expression to filter a graphical query

- 3 To indicate that an ad hoc expression completes the column, select Ad-Hoc.
- 4 If necessary, choose Ellipsis to change the parameter properties, such as data type, display name, display length, and display style. You can also specify whether a parameter is required or hidden.

Filtering a textual query using a user-supplied expression

When you use an ad hoc parameter to allow or require a user to specify conditions that returned records must meet, you identify the name of a column, and the user supplies the condition for that column.

To use an ad hoc parameter in the SQL code for a textual query, complete the following tasks in this order:

- Include an ad hoc parameter in the query.
- Prepare the query to describe the parameter.
- Modify the property values of the parameter, if necessary.

Including an ad hoc parameter in a textual query

You can specify that e.Report Designer Professional insert the conditional expression for an ad hoc parameter in a SQL SELECT statement.

Examples of the required syntax follow:

```
<space>:?ad hoc_param<space>
```

or:

```
<space>:?ad hoc_param<end-of-statement>
```

You must use a leading space. The colon (:) and question mark (?) indicate an ad hoc parameter. The name of the ad hoc parameter is `ad hoc_param`. A space follows an ad hoc parameter name, unless the parameter name is the last word in the SQL SELECT statement.

For example, the following SQL query specifies an ad hoc parameter that is called `creditrank`:

```
SELECT customname, creditrank from customers WHERE :?creditrank
```

Typically, the ad hoc parameter name is the same as the name of the column for which the user can specify a value. Parameter names are not case-sensitive. An ad hoc parameter cannot have the same name as a static parameter.

Preparing a textual query to describe an ad hoc parameter

When you choose Describe Query to prepare and describe the query, Actuate e.Report Designer Professional provides information about the ad hoc parameter on Textual Query Editor—Adhoc Parameters.

Modifying the property values of the ad hoc parameter



You can set the property values for an ad hoc parameter on Parameter Properties by choosing Ellipsis next to the ad hoc parameter on Textual Query Editor—Adhoc Conditions. You also can set the associated column name for an ad hoc parameter on Textual Query Editor—Adhoc Conditions. By default, e.Report Designer Professional fills in the parameter name as the name of the associated column. If you use a parameter name that is not the same as the name of the associated column, modify the value in Column Name appropriately.

Figure 9-6 shows a SQL SELECT query that specifies an ad hoc parameter. The lower part of the window displays the resulting values that appear on Textual Query Editor—Adhoc Conditions.

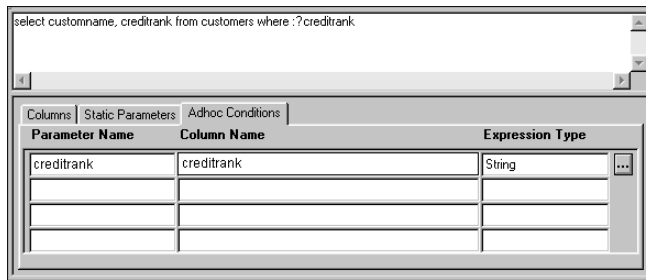


Figure 9-6 A SQL SELECT query that specifies an ad hoc parameter

For information about using Parameter Properties, see *Developing Reports using e.Report Designer Professional*.

How to change the name and data type of the column that is associated with an ad hoc parameter

On Textual Query Editor—Adhoc Conditions:

- In Column Name, specify the column name to use with the ad hoc parameter. By default, the column name is the same as the parameter name.
- In Expression Type, specify the Actuate Basic type of the column, as shown in Figure 9-7.

e.Report Designer Professional uses these values to generate the appropriate syntax for the conditional expression that includes the ad hoc parameter.

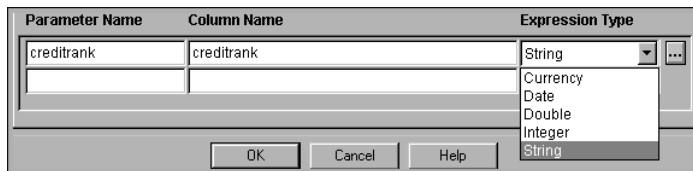


Figure 9-7 Selecting the Actuate Basic type of a column

10

Maintaining a database or ODBC query

This chapter contains the following topics:

- About maintaining database and ODBC queries
- Timing and optimizing data source queries
- Updating a query when the data source changes

About maintaining database and ODBC queries

Typically, after you create a database or ODBC query for a report design, you do not need to change it. You must, however, maintain the query if the data source structure changes. If your data source administrator deletes or renames tables and columns change, you must update your query to match. You can have e.Report Designer Professional determine whether the tables and columns have changed in the data source and synchronize your query to the modified data source.

If you find that you need faster query performance for an existing query, you can time and tune the query.

Timing and optimizing data source queries

When you preview the data that a database query returns, the status bar on the bottom left lists performance statistics. For long-running queries, you can use these statistics to help tune your query.

You also copy the SQL statement your query sends to the data source and test its performance in a utility that your data source provides. Using the performance data and your knowledge of SQL, you can determine if you can optimize the SQL statement to provide the required results more efficiently. You can then modify the SQL statement in your query. To use this technique, you need to determine what SQL statement is sent to the data source. When you use a graphical query, you can view most of the information that you need on the SQL page of Query Editor. When you use a textual query, you type the SQL statement, but various features, such as parameters and sort keys, can modify the SQL statement.

Viewing the SQL SELECT statement that e.Report Designer Professional sends to the data source

Sort keys and conditions can modify the SQL code that appears in the upper part of Textual Query Editor or on Query Editor—SQL before the query is executed.

You can obtain the SQL SELECT statement that is sent to the data source by using the `ObtainSelectStatement()` method of the data source. For a graphical query, this method is part of the `AcSqlQuerySource` class. For a textual query, this method is part of the `AcTextQuerySource` class. For more information about `AcSqlQuerySource` and `AcTextQuerySource`, see *Programming with Actuate Foundation Classes*.

How to view the complete SQL SELECT statement that e.Report Designer Professional sends to the data source

- 1 In Report Structure, right-click the data stream component, and choose Properties.
- 2 On the Properties page for the component, choose Methods.
- 3 On the Methods page for the component, select Function ObtainSelectStatement() As String.



- 4 Choose Edit.

Method Editor appears, displaying the following code:

```
Function ObtainSelectStatement() As String
    ObtainSelectStatement = Super::ObtainSelectStatement( )
End Function
```

- 5 Above the following line:

```
End Function
```

type:

```
ShowFactoryStatus(ObtainSelectStatement)
```

- 6 Choose Close.
- 7 On the Methods page for the component, choose Close.
- 8 Choose Report→Build and Run.
- 9 On Requester, type desired values for any parameter.
Choose OK.

The Report Viewer contains the Actuate Output which has the SQL SELECT statement that e.Report Designer Professional sent to the data source. You can copy this statement to test and tune it in your data source.

Modifying the SQL code in a query to optimize the query

After you extract the SQL SELECT statement from a query and tune it in your data source, you can incorporate the desired changes in the query in e.Report Designer Professional.

If you use a textual query, you change the SQL code directly. You can also change the sorting options. If you want to programmatically override the statement that you input in Textual Query Editor, select the data source, and change the SelectStatement property of the AcTextQuerySource class. You can also modify the SetUpAdHocCondition() of the class to specify additional conditions for the query. For more information about AcTextQuerySource, see *Programming with Actuate Foundation Classes*.

If you use a graphical query, you can modify the performance by performing one or more of the following actions:

- Modify the FROM clause of the query.
- Modify the conditions and summarization to change the WHERE, GROUP BY, and HAVING clauses.
- Modify the sorting to change the ORDER BY clause.
- Modify the effect of group section sort keys on the ORDER BY Clause.
- Convert a graphical query to a textual query, and edit it to include the desired optimizations.
- Override the ObtainSelectStatement method of the data stream to insert your own SQL statement. To override this method, replace the call to `Super::ObtainSelectStatement( )` in the method with your own code. If you want to override only particular clauses, Table 10-1 describes the variables that you can override in `AcSQLQuerySource`.

Table 10-1 Variables that you can override in `AcSQLQuerySource`

Variable	SQL SELECT clause
Dim SelectClause As String	SELECT clause
Dim FromClause As String	FROM clause
Dim WhereClause As String	WHERE clause
Dim GroupByClause As String	GROUP BY clause
Dim HavingClause As String	HAVING clause
Dim OrderByClause As String	ORDER BY clause

Updating a query when the data source changes

If the data source changes, an existing query can refer to tables or columns that no longer exist or have changed. If you know that the data source changed, you can check a query and modify it as needed. Otherwise, you can discover and address the situation when the query produces a data source error. You also can send SQL statements from e.Report Designer Professional to update the database or other ODBC data source.

If you use a textual query, you must know what changes to make to synchronize the query with the data source. With a textual query, synchronize only the items to which the query refers.

If you use a graphical query, you can request that e.Report Designer Professional synchronize the query. e.Report Designer Professional synchronizes all tables and columns in the data source, even if the graphical query only uses a few of them.

Sending SQL statements to change the data source

You can create and send your own SQL statements to a database or ODBC data source. For example, you can send a SQL statement to update a record, drop a table, and so on by calling the statement's `Execute()` method after `Prepare()`. Use the statement's `Execute()` method only to execute a statement that does not return data rows, such as `UPDATE`, `DELETE`, or `DROP TABLE`. To execute a SQL statement that returns rows, use a cursor.

How to execute a SQL statement that does not require a cursor

- 1 Establish the connection.
- 2 Prepare the SQL statement using the connection's `Prepare()` method.
`Prepare()` returns a reference to the DB Statement object.
- 3 If the SQL statement accepts parameters whose values are provided later, use the statement's `BindParameter()` method to bind the parameters to the values.
- 4 Execute the statement using the statement's `Execute()` method.
`Execute()` returns `True` or `False`, depending on whether the statement runs successfully.
- 5 The Factory or custom code deletes the statement.
- 6 The Factory or custom code deletes the connection.

For more information about Actuate Foundation classes and the Factory, see *Programming with Actuate Foundation Classes*.

Updating a textual query when the data source changes

The only data source changes that affect a textual query are changes to the tables and columns that the SQL statement in the textual query uses. In a textual query, e.Report Designer Professional does not store additional information about the data source tables and columns.

Table 10-2 specifies the type of change to make to synchronize a textual query after a data source change.

Table 10-2 Types of changes required to synchronize a textual query with a changed data source

Object	Change	Query synchronization
Table	Name	Change the name of the table wherever it appears in the query.
	Join columns	Change the WHERE clause appropriately.

(continues)

Table 10-2 Types of changes required to synchronize a textual query with a changed data source (continued)

Object	Change	Query synchronization
Column	Name	Change the name of the column wherever it appears in the query.
	Datatype	Make all other synchronization changes first, then describe the query. Next, change the Actuate data type on Textual Query Editor—Column. If your query uses a static parameter to specify the value for a parameter, change the Actuate data type of the static parameter on Textual Query Editor—Static Parameters. If your query uses an ad hoc parameter to specify the value for a parameter, change the Actuate data type of the ad hoc parameter on Textual Query Editor—Adhoc Conditions.

Updating a graphical query when the data source changes

A report object design (.rod) file or report object library (.rol) file that has a graphical query stores the relevant structure of the data source, which is known as the database schema or data dictionary. The database schema evolves as the database design evolves. Therefore, a query in a report object design that originally addressed a certain set of tables and views in a data source can become out of sync with the database schema, and the data source to which it connects. e.Report Designer Professional displays and must synchronize all the tables and columns in the data source, even if the SQL SELECT statement that is generated by the graphical query uses only a few of them.

The query synchronization tools in e.Report Designer Professional help you work with the problem of data source changes by:

- Displaying and analyzing the differences between the tables, views and columns that are defined in the query and those in the data source. You can add, delete or rename tables, views and columns. You can also change the column data type.
- Modifying tables, views, and columns to synchronize the query to the data source. Synchronization preserves all conditions, sorting, and other information in a query. The query synchronization process contains several updating features. For example, you can choose to accept an update, which adds new columns and changes column data types as necessary.

- Verifying that a SQL statement has valid syntax. You also can choose to verify a SQL statement without synchronizing the query.

Synchronization does not change:

- The data type of a control that the report uses to display a column. e.Report Designer Professional's Factory checks for the compatibility of data columns and controls at report build time. The build process also checks for incompatible or deleted column types.
- Customized and developer-defined methods. Synchronization does not report or update customized or developer-defined methods that refer to a parameter or row variable that changed as a result of synchronization.

When you synchronize a query, the following steps occur in this order:

- Making a backup copy of your report file. Your report changes when you accept updates during synchronization. Consider making a backup copy of your file before you begin the synchronization process in case you want to undo any of your changes.
- Ensuring that a query connects to the correct data source.
- Updating the data dictionary during synchronization.
- Determining whether a graphical query and its data source are synchronized.
- Synchronizing a graphical query.
- Verifying that synchronization is complete.

Ensuring that a query connects to the correct data source

When you work with several versions of a data source, you can lose track of the one to which you are connected. For query synchronization to work effectively, the report design must refer to the correct data source. Ensure that you connect to the correct data source before you begin synchronization.

Some databases, such as Microsoft Access, are accessed using a data file. However, most databases, such as Oracle, are accessed using a connection to a database client. You connect to and install a database or ODBC data source that you access using a file differently from one that you access using a client.

If you access the database or ODBC data source using a client, Query Editor displays the qualified data source name, such as AcTestDB, in Database Browser, as shown in Figure 10-1.

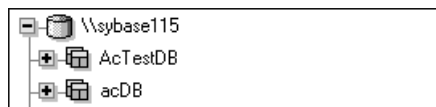


Figure 10-1 Database Browser, displaying the qualified data source name

How to ensure that you connect to the correct data source that you access using a client

- 1 In Query Editor, right-click a table, and choose Properties.
- 2 In Query Editor—Table Property, change the qualifier to the correct database or ODBC data source, if necessary.
- 3 Choose Apply.

With a file-based data source, such as MS Access, ensure that the full path names match if you installed your version of e.Report Designer Professional in a nonstandard location, or are unsure of the full path names of the database or ODBC data sources with which you are working.

How to ensure that you connect to the correct file-based data source

- 1 In Query Editor, right-click a table, and choose Properties.

Query Editor—Table Property appears.

Compare the path name that appears at the top of Database Browser with the qualified path name that appears on Query Editor—Table Property. Database Browser displays the name of the connected data source and its installed location, as shown in Figure 10-2.

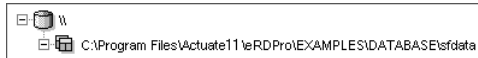


Figure 10-2 Database Browser, displaying the full path of the connected data source

- 2 If these path names do not match, complete the following steps:
 - 1 On Query Editor—Table Property, change the file path to the correct path of the database or ODBC data source to which you want to connect, as shown in Figure 10-3.

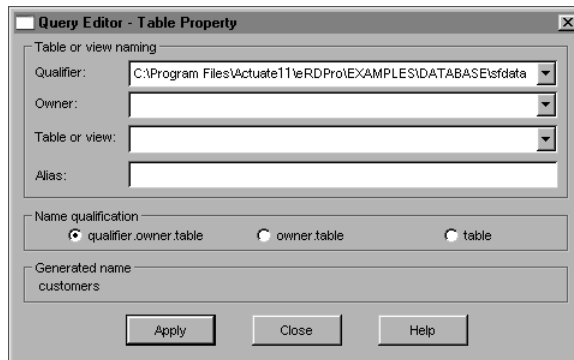


Figure 10-3 Changing the file path

- 2 Choose Apply to initiate the table consistency check.

- 3 Choose Close.

Updating the data dictionary during synchronization

Query synchronization examines the data dictionary that e.Report Designer Professional stores with the report object design (.rod) or report object library (.rol) file. The data dictionary is basically a copy of the database schema for the connected database or ODBC data source. The database schema of the connected data source can change while Query Editor is open. In this case, you must refresh the data dictionary with the database schema information. Refreshing the data dictionary while checking a query ensures that you have the correct data dictionary information.

How to set synchronization to refresh the data dictionary

- 1 In Query Editor, choose Tools→Options.
Options—Design Editor appears.
- 2 Choose Query Editor.
Options—Query Editor appears.
- 3 Select Refresh Data Dictionary when checking a query, as shown in Figure 10-4.

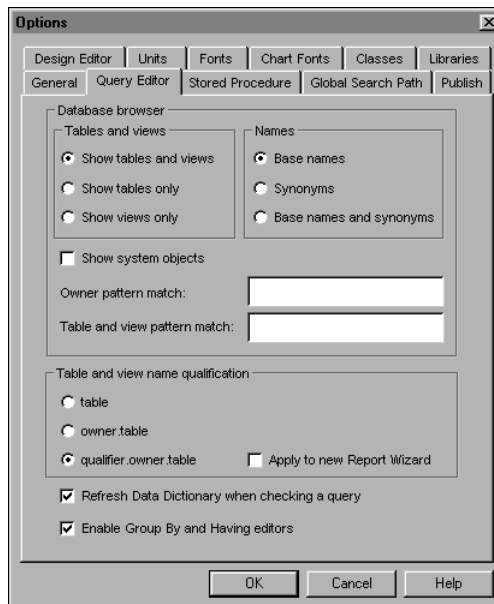


Figure 10-4 Indicating that you want e.Report Designer Professional to refresh the data dictionary when checking a query

Choose OK.

Determining whether a graphical query and its data source are synchronized

When you synchronize a query, e.Report Designer Professional reports any differences between the database schema and the graphical query's information about the database schema.

How to determine whether a graphical query is synchronized

- 1 In Query Editor, choose SQL→Synchronize Query.

The results depend on whether the query is synchronized:

- If the query is not synchronized, Synchronize Query With Schema appears.
- If the query is synchronized, e.Report Designer Professional displays the following message:

The query definition is synchronized with the database schema.

- 2 If the query is synchronized with the data source:

- 1 Choose OK.

If the query creates a valid SQL statement, e.Report Designer Professional displays the following message:

Statement is valid.

- 2 Choose OK.

- 3 If the query is not synchronized, Synchronize Query With Schema appears, as shown in Figure 10-5. On Synchronize Query With Schema:

- 1 Examine the list of tables from the FROM clause of the query that no longer appear in the database schema.
- 2 In Columns, examine the fully qualified names of tables and their columns that have changed. If a table has an alias, the table appears under each of its aliases.

Columns also displays the type of change that e.Report Designer Professional detects for the table or column during synchronization. The Old DB type column displays the column data type that is stored in the query, and the New DB Type column displays the data type that was detected in the database or ODBC data source.

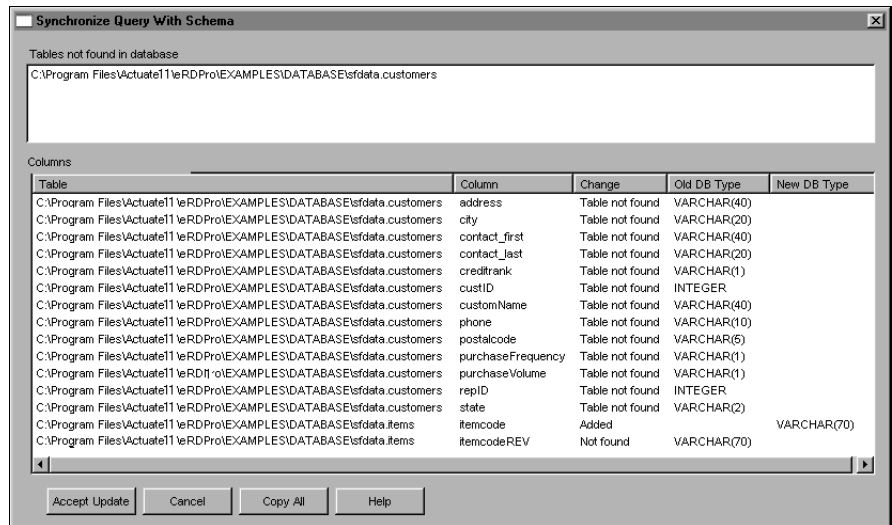


Figure 10-5 Synchronization error messages

Synchronizing a graphical query

If Synchronize Query With Schema reports synchronization problems, complete the following steps to update the tables, columns, and views in a graphical query:

- On Synchronize Query With Schema, choose Accept Update to update the new table definitions.

Columns that are not synchronized are highlighted and labeled in the upper part of the query editor, as shown in Figure 10-6.

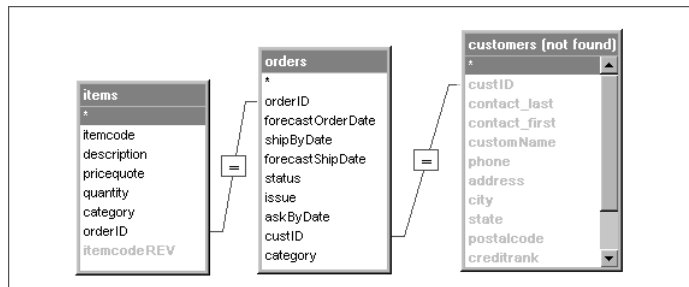


Figure 10-6 Query Editor, highlighting an unsynchronized column in the items table and all columns in customers, a missing table

Information about the following synchronization issues appears on Actuate Output, as shown in Figure 10-7:

- Items to which the query refers that have been modified
- Tables and columns that are not found but to which the query refers

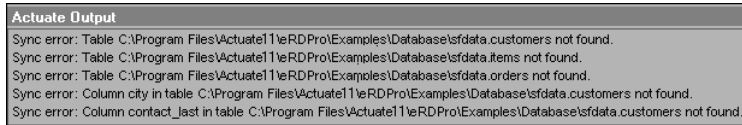


Figure 10-7 Actuate Output, showing synchronization issues

You can use the content of Actuate Output as a checklist to track the changes that you make to the query.

- Double-click an item in Actuate Output. Either a message appears and advises you how to proceed, or the appropriate tool to make the necessary change appears, as described in the following list:
- Table invalid or not found

Double-clicking this type of entry opens Query Editor—Table Property, as shown in Figure 10-8. On Query Editor—Table Property, rename the table in the query to match the table name in the database or ODBC data source.

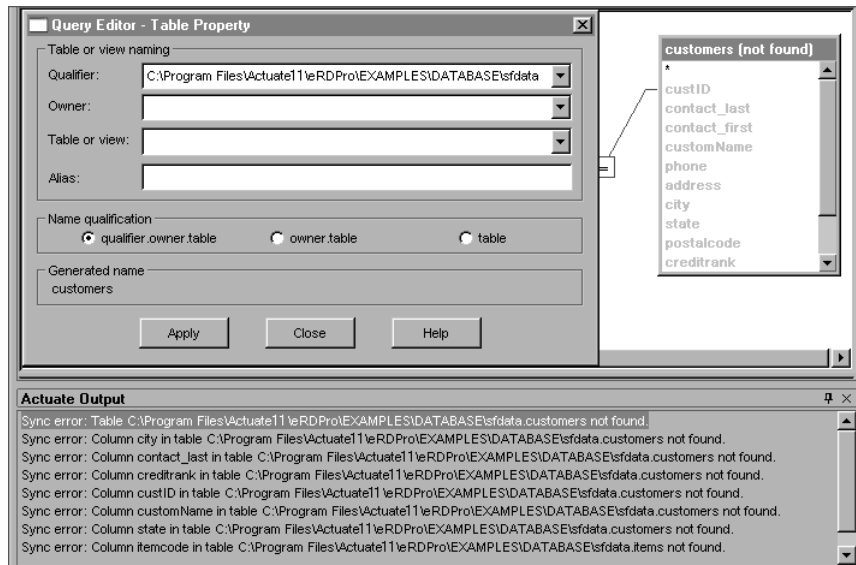


Figure 10-8 Table properties

If the table is obsolete and does not need to be replaced, go to the table's context menu, as shown in Figure 10-9, and remove the table.

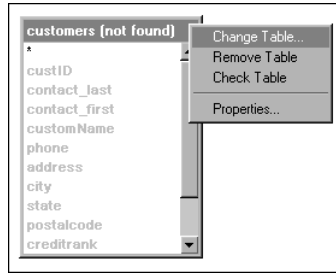


Figure 10-9 The table's context menu

- **Column not found**

Double-clicking this type of entry opens Change Column Used in Query, as shown in Figure 10-10. On Query Editor—Change Column Used in Query, change the original column reference to a valid reference, or remove all references to the column from the query.

In the New column, a drop-down list displays all available valid names in the stored table definition of the data dictionary. Changing the column name fixes the query by replacing the original column name with the new column name. The new column name replaces the old column name on Query Editor—Columns and in the upper part of Query Editor.

Removing column references removes all references to the original column in the query, including from the query statement.

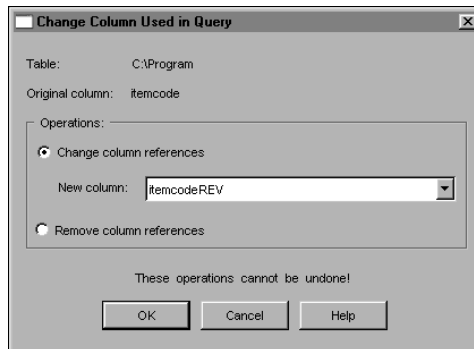


Figure 10-10 Change Column Used in Query

If the table that contains the problem column no longer exists in the database or ODBC data source, resolve the table problem before you change the column. Figure 10-11 shows sample output for when a table and its columns do not exist in the database or ODBC data source.

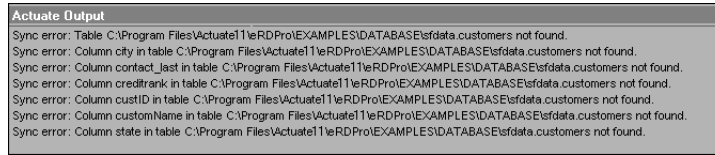


Figure 10-11 Actuate Output, indicating that a table is missing

- Other issues

If a message appears instead of a tool, follow the instructions in the message to resolve the synchronization issue. For example, if e.Report Designer Professional cannot find a particular column, it displays a message like the following message:

```
The column Year is defined in table C:\Projects\Chart
  \Examples\Database\ChartExamples.RegionalSales which
  cannot be found. Change the table property to reference
  the appropriate table or remove the table from the query.
```

Verifying that synchronization is complete

After you complete the changes to the columns, tables, and views to eliminate synchronization errors, complete the following steps:

- Verify that no tables or columns are highlighted in Query Editor.
Highlighting in Query Editor indicates that synchronization issues exist for the table or column.
- Choose SQL→Synchronize Query.
If the query's stored table definition is synchronized with the data source, e.Report Designer Professional displays the following message:

```
The query definition is synchronized with the database schema.
```

If the query is synchronized, the synchronized processes verify that the SQL statement syntax is valid. e.Report Designer Professional passes SQL syntax verification to the connected database or ODBC data source server for processing. The server's scope of syntax checking varies by vendor. If the SQL statement is not valid, e.Report Designer Professional displays an appropriate error message. When checking is complete, e.Report Designer Professional displays the following message:

```
Statement is valid.
```

Accessing data using a stored procedure

This chapter contains the following topics:

- About accessing a database using a stored procedure
- Preparing to use a stored procedure to access a database
- Designing a report that uses a stored procedure
- Selecting a stored procedure
- Working with data from a stored procedure
- Using parameters with stored procedures
- Ensuring that the stored procedure is synchronized with the database
- Using Oracle stored procedures
- Customizing access to handle specific databases or multiple result sets

About accessing a database using a stored procedure

To access data from a database, you use a query or a stored procedure. A stored procedure is a block of SQL statements that perform a specific task. An Actuate report can call a stored procedure in a database that supports stored procedures. Use of a stored procedure improves data access performance by reducing the amount of information that is sent over a network. You can use a stored procedure to execute any database task, such as modifying, inserting, or deleting records or exchanging data between the database and an Actuate report.

Actuate e.Report Designer Professional supports the automated stored procedure features of the following databases:

- Databases that are accessed through ODBC, including Oracle, PeopleSoft, and Progress
- Oracle9i, Oracle 9.2, or Oracle 10g R1 databases that are used with an Oracle9i client
- IBM DB2 databases

To access a database using a stored procedure, complete the following tasks in this order:

- Create a report that accesses a stored procedure data source.
- Select a stored procedure.
- Place data from the stored procedure in the report design.
- If your procedure uses parameters, specify the type of parameter or specify a sample value, if necessary.
- As part of the general maintenance of your report, periodically synchronize your stored procedure to verify that the definition of the stored procedure has not changed in the database.

You can work with a stored procedure by using the stored procedure tools in e.Report Designer Professional or by overriding an Actuate Basic method to customize accessing a stored procedure. This chapter discusses both techniques.

To access a stored procedure that returns a single result set, use Stored Procedure Data Source Builder. To process multiple result sets, you must override a method that modifies the SQL statement.

Preparing to use a stored procedure to access a database

Before using a stored procedure to access a database, you must:

- Provide the database or ODBC connection.
To access data using a stored procedure or SQL query, e.Report Designer Professional requires the computer to have a database connection or a connection to an ODBC data source.
- Define a database connection component.
Define a database connection component in e.Report Designer Professional to use the database or ODBC connection.

You need the following information:

- The name of the data source
- User name and password for the database login
- Name of the stored procedure
- Names of the stored procedure's result fields that the report requires

With some types of databases, you also need the following information:

- Whether a parameter is an input or output parameter
- Whether the procedure is a user procedure or a system procedure, if you want to limit the procedures that are available for selection

Designing a report that uses a stored procedure

A report that uses a stored procedure requires a connection to a data source that stores procedures. It also requires the `StoredProcedureSource` data stream. You can place this data stream in an existing report or you can insert it when you create a blank report.

How to design a report that uses a stored procedure

To build a new report that accesses and works with a stored procedure:

- 1 In e.Report Designer Professional, choose **File**→**New**.
- 2 In **Create New Report**, select **Blank Report**. Choose **OK**.
- 3 Select the connection component, as shown in Figure 11-1.



Figure 11-1 The connection component

Right-click, and choose Properties.

Properties—Properties displays the properties for the connection.

- 4 Set the database connection properties, as shown in Figure 11-2. The type of database determines which properties are required. Choose OK.

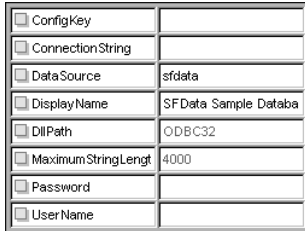


Figure 11-2 Setting the database connection properties

- 5 In Report Structure, right-click DataStream. Choose Delete.



- 6 Drag a database source component from Toolbox—Data, and drop it into the empty DataStream slot, as shown in Figure 11-3.

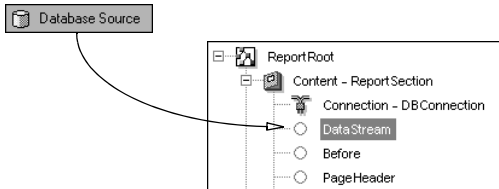


Figure 11-3 Adding a database source component

- 7 In Select Component, select Stored Procedure Data Source, as shown in Figure 11-4.

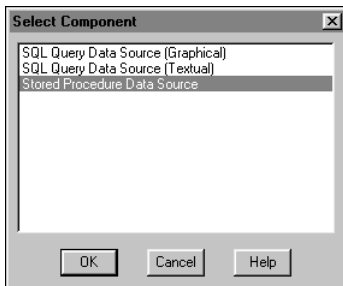


Figure 11-4 Selecting Stored Procedure Data Source

Choose OK.

The stored procedure data source appears in the report design, as shown in Figure 11-5. This component connects to a data source that contains a stored procedure. You can add this component to an existing report or insert it in the report design when you build a blank report.

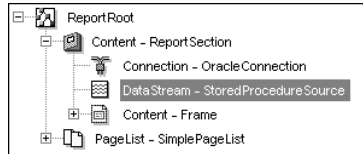


Figure 11-5 Report Structure, showing the stored procedure data source

You are now ready to select a stored procedure to use in your report.

Selecting a stored procedure

After you set up the report design, you can select a stored procedure to create a stored procedure component.

How to select a stored procedure

- 1 In e.Report Designer Professional, right-click DataStream—StoredProcedureSource, and choose Data Source.

Database Login appears if the data source requires login information.

- 2 Type any required information, as shown in Figure 11-6.

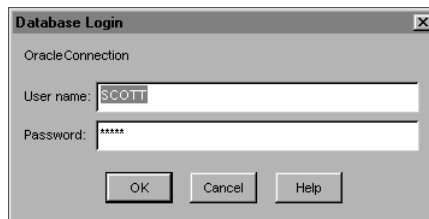


Figure 11-6 Providing login information

Choose OK.

Stored Procedure Data Source Builder appears. If the database contains multiple stored procedures, Stored Procedure Data Source is blank, and Stored Procedure Browser also opens, as shown in Figure 11-7.

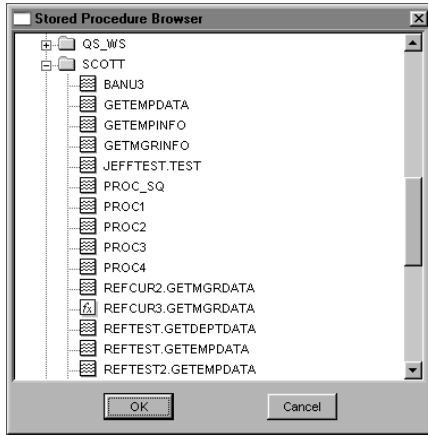


Figure 11-7 Stored Procedure Browser

You also can access Stored Procedure Browser by right-clicking `DataStream—StoredProcedureSource` and choosing `Sample Parameter Values` when `Stored Procedure Data Source Builder` is open.

- 3 If `Stored Procedure Browser` appears, double-click the name of the stored procedure that you want to use.

If the stored procedure uses parameters, `Sample Parameter Values` appears, as shown in Figure 11-8. `Sample Parameter Values` supports using a sample input value for a parameter with a stored procedure. You also can access this tool by right-clicking `DataStream—StoredProcedureSource` and choosing `Sample Parameter Values` when `Stored Procedure Data Source Builder` is open.

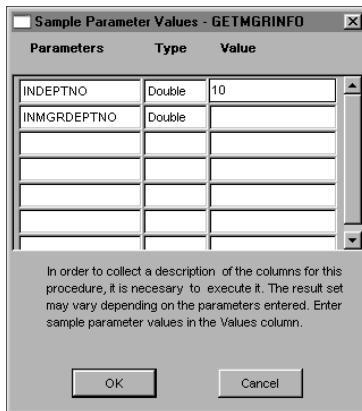


Figure 11-8 Sample Parameter Values

- 4 If `Sample Parameter Values` appears, choose `OK`.

The procedure appears in Stored Procedure Data Source Builder. This tool supports viewing and modifying details about the stored procedure.

In Figure 11-9, the stored procedure produces information that appears on the Result Columns page. Some stored procedures, however, are internal functions that do not return data to users.

The screenshot shows the 'Stored Procedure Data Source Builder' dialog box. At the top, the 'Name' field contains 'SCOTT.GETMGRINFO' and there is a 'Procedures...' button. Below this, the 'Procedure name qualification' section has three radio buttons: 'name' (selected), 'owner.name', and 'db.owner.name'. The 'Return type' field is empty. The main area has three tabs: 'Result Columns', 'Parameters', and 'Description'. The 'Result Columns' tab is active, showing a table with two columns: 'Column Name' and 'Actuate Type'. The table contains two rows: 'ENAME' with 'String' and 'DEPTNO' with 'Double'. There are empty rows below.

Column Name	Actuate Type
ENAME	String
DEPTNO	Double

Figure 11-9 Result Columns

If any of the stored procedure's result columns have the same name, Actuate e.Report Designer Professional adds a suffix to each duplicate column name so that every column name is unique. For example, if two result columns have the name MYCOL, e.Report Designer Professional renames one of them MYCOL_1.

Changing the name of a stored procedure component



Stored Procedure Name Editor, as shown in Figure 11-10, supports modifying the name of the stored procedure component. You access this tool by right-clicking DataStream—StoredProcedureSource and choosing Stored Procedure Name Editor when Stored Procedure Data Source Builder is open.

The screenshot shows the 'Stored Procedure Name Editor' dialog box. It has three text fields: 'Qualifier' with 'SCOTT', 'Owner' (empty), and 'Procedure' with 'GETMGRINFO'. At the bottom are 'OK' and 'Cancel' buttons.

Figure 11-10 Stored Procedure Name Editor

If you edit the name, ensure that the stored procedure's parameter and result columns are consistent with the new name.

Limiting which stored procedures are available for selection

When you select a stored procedure, you can select from a list of some or all of the stored procedures. A feature that is available on Options—Stored Procedure supports narrowing the list of stored procedures.

How to limit which stored procedures appear in the selection list

- 1 Choose Tools→Options→Stored Procedure, as shown in Figure 11-11.

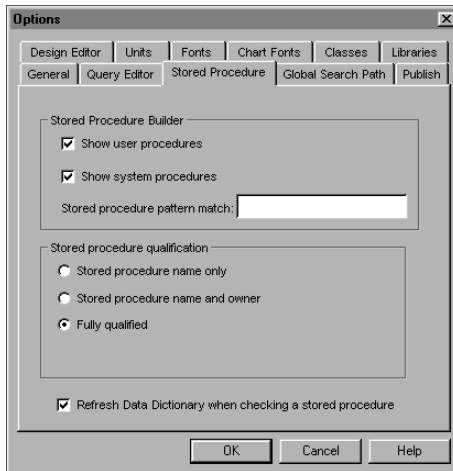


Figure 11-11 Options—Stored Procedure

- 2 On Stored Procedure, indicate whether you want to view and work with user procedures, system procedures, or both.

If your database cannot distinguish between user procedures and system procedures, these settings have no effect, and all stored procedures appear for selection.

- 3 To narrow the list of stored procedures, type a search expression for Stored procedure pattern match, such as:

"*Salary*"

Using this expression, Stored Procedure Browser lists only stored procedures with Salary in the name.

Working with data from a stored procedure

After you build a report, access the stored procedure data source, and select a stored procedure, you can add data from the stored procedure to your report layout.

How to place data from a stored procedure in a report design



- 1 Select the frame in the Content slot of your report design to make it active.
- 2 Choose View→Fields.

The controls from the stored procedure are now available to drag from the list and drop in the frame of the report design.

The list of controls in Fields matches the list in Result Columns in Stored Procedure Data Source Builder. Fields displays the controls in alphabetical order, as shown in Figure 11-12. Result Columns displays them in the order used in the stored procedure definition.

- 3 From Fields, drag the items that you want to appear in the report, and drop them in the Content frame.

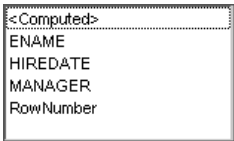


Figure 11-12 Fields

- 4 Arrange controls in the frames. You typically place column headers and controls for totals in Before and After frames, as shown in Figure 11-13.

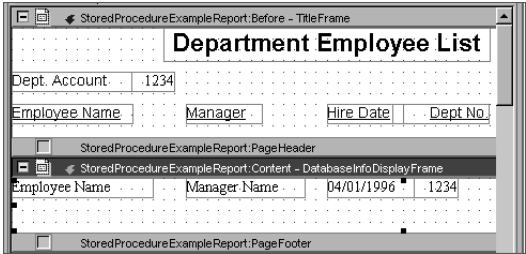


Figure 11-13 Arranging controls in the frames

- 5 Choose Report→Build and Run.

If the stored procedure uses parameters, e.Report Designer Professional prompts the user to type an input parameter value, as shown in Figure 11-14. In this example, 3 is the value of the input parameter. After you supply the necessary input parameter values, choose OK.

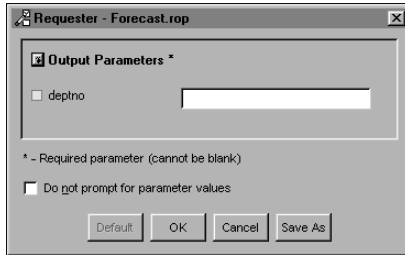


Figure 11-14 Requester

The report's output is shown in Figure 11-15.

Department Employee List			
Dept. Account	120		
<u>Employee Name</u>	<u>Manager</u>	<u>Hire Date</u>	<u>Dept No.</u>
ADAMS	SCOTT	05/23/1987	20
FORD	JONES	12/03/1981	20

Figure 11-15 The report output

Using parameters with stored procedures

Stored procedures can use input and output parameters. If your stored procedure uses parameters, you may need to perform the following tasks:

- Specifying whether parameters are input or output parameters
- Working with a sample value for an input parameter

Specifying whether parameters are input or output parameters

If you use ODBC, Actuate software sets parameters to the correct type, either input or output. Some vendors' database systems do not provide information about whether a stored procedure parameter is an input or output type. If this is the case, UNKNOWN appears beside the parameter name on Stored Procedure Data Source Builder—Parameters, under Input/Output. If UNKNOWN appears, you must designate the parameter type in Stored Procedure Data Source Builder.

The parameter's type determines how e.Report Designer Professional handles the variable for that parameter:

- e.Report Designer Professional generates the name of an input parameter variable. Typically, the name of the variable is the same as the parameter name

in the stored procedure. Then, e.Report Designer Professional attempts to match the name of the input parameter variable to either a local, inherited or global parameter that you defined earlier.

If there is no parameter definition, e.Report Designer Professional defines a local parameter. If the name of the input parameter variable conflicts with the name of a variable that is not a parameter, e.Report Designer Professional appends an underscore and a number to the name, such as MYPARAM_1.

- e.Report Designer Professional stores an output parameter as a variable on the stored procedure component.

How to designate a parameter type

- 1 In Stored Procedure Data Source Builder, choose Parameters, and review the data.
- 2 In the Input and Output column for parameters that have an Input and Output designation of UNKNOWN, indicate whether the parameter is an input parameter, an output parameter, or both, as shown in Figure 11-16.

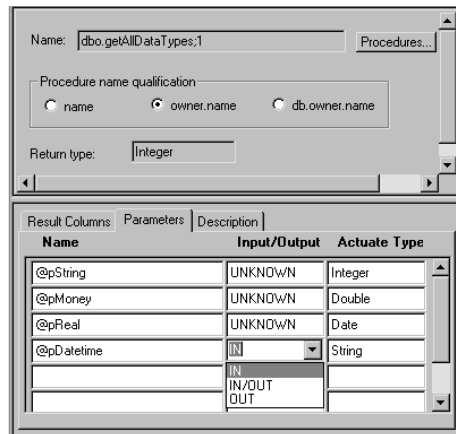


Figure 11-16 Indicating whether a parameter is an input parameter, output parameter, or both

The named field items that you specify as input parameters appear on Requester when a user runs the report, as shown in Figure 11-17.



Figure 11-17 Requester, showing the input parameters

Working with a sample value for an input parameter

To obtain and display information about the columns, Stored Procedure Data Source executes the stored procedure. Sometimes, a stored procedure returns different types of result sets according to input parameter values. To get the appropriate result set, Stored Procedure Data Source Builder displays a prompt for a parameter's values if the stored procedure requires input parameter values.

How to specify a sample value for an input parameter

- 1 To determine whether the stored procedure uses parameters, choose Parameters in Stored Procedure Data Source Builder, as shown in Figure 11-18.

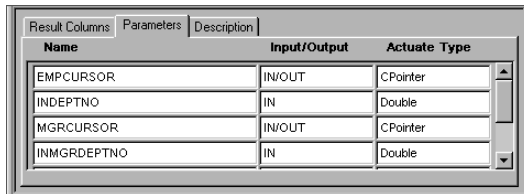


Figure 11-18 Parameters for the stored procedure

If the report uses or requires parameters from the stored procedure, the following information is available on Stored Procedure Data Source Builder—Parameters:

- The name of the field from the stored procedure that uses a parameter
- Whether the parameter is an input or an output parameter
- The Actuate Basic type of the parameter, which can be any of the following types:
 - CPointer
 - Currency
 - Double
 - Date
 - Integer
 - String

If e.Report Designer Professional requires an input value for a stored procedure to determine the result column, e.Report Designer Professional displays Sample Parameter Values to enable you to specify that input value. If Sample Parameter Values appears, as shown in Figure 11-19, type a parameter value, such as 3, in the Value column for each parameter. Then choose OK.

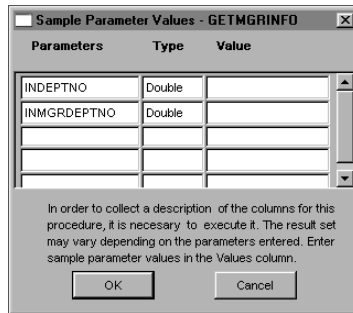


Figure 11-19 Sample Parameter Values

Setting a Sybase Stored Procedure parameter to an empty string

If you are using Sybase and are unable to set a Sybase Stored Procedure parameter to an empty string, you must select Enable Describe Parameter in the ODBC data source, as shown in Figure 11-20.

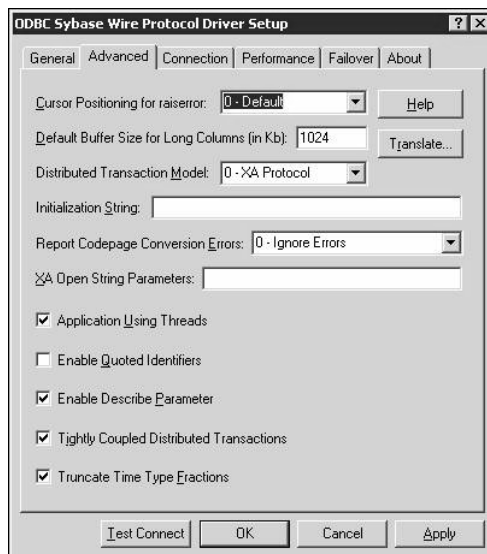


Figure 11-20 Setting Enable Describe Parameter

If you are using a UNIX or Linux based operating system, add the following line to the DSN entry in the odbc.ini file:

```
EnableDescribeParam=1
```

If you are using the ConnectionString property in a report, you may add the EnableDescribeParam=1 clause to the property string.

Ensuring that the stored procedure is synchronized with the database

A stored procedure in your database can change between the time of the original report design and when you run it in a report. It is a good practice to verify that the stored procedure that you use is current. Use Check Stored Procedure to determine whether the content of a stored procedure in your design is consistent with its definition in the database.

To verify whether a stored procedure is synchronized, edit the data source component. If prompted, log into the database. Select the stored procedure to verify and then choose SQL→Synchronize Stored Procedure. If the stored procedure uses input parameters, Sample Parameter Values appears so you can specify values for the input parameters.

If the stored procedure does not use parameters, a message appears, indicating whether the stored procedure definition is consistent with the database's definition of the stored procedure. appears. A message appears, indicating whether the stored procedure definition in e.Report Designer Professional is consistent with the database's definition of the stored procedure.

Using Oracle stored procedures

Stored Procedure Data Source Builder supports Oracle stored procedures and stored functions. Do not use Oracle stored procedures and stored functions that return parameters with Boolean, indexed table, or record data types. Oracle Call Interface does not support returning parameters with those data types. You must execute a stored procedure or stored function to determine the columns in the result set. To execute a stored procedure or stored function, enter sample values for the input parameters on Sample Parameter Values.

In some cases, a stored procedure or stored function returns different columns depending on the values of the input parameters and how the stored procedure or stored function is defined. The columns in the result set, therefore, might not correspond to the controls in your report design. Stored Procedure Data Source Builder supports a weak cursor-type definition if it always returns the same set of result columns. A weak cursor supports determining the return type when the report runs.

Using Oracle data types

For Oracle stored procedures, Actuate products support retrieving NCHAR and NVARCHAR2 data types with the following limitations:

- NCHAR support includes the use of NCHAR data types in the result set but not for input, output, or input and output parameters.
- You cannot use static or ad hoc parameters or conditions on an NCHAR database column.
- Actuate supports NCHAR data types in its native Oracle DBMS module, not in its ODBC DBMS module.

You can use these data types with alternate character sets. They map to the Actuate String data type.

Accessing stored functions

A stored function is a stored procedure that returns a function value. The returned value can have any data type that Oracle supports, and it can be a cursor variable. If a stored function returns a cursor variable, Stored Procedure Data Source Builder can process the result set.

Identifying stored procedures and stored functions



The Stored Procedure Browser displays the type and scope of each stored procedure and stored function. A stored procedures and functions can be part of a packaged procedure. The full name appears in Stored Procedure Browser as qualifier.package.procedurename, as shown in Figure 11-21.

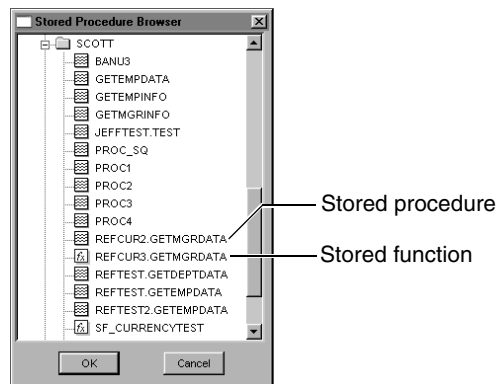


Figure 11-21 Stored procedures and stored functions

The Oracle schema maps to the Actuate qualifier.

Processing additional cursors that a procedure or function returns

Stored Procedure Data Source Builder supports only stored procedures and stored functions that return one result set, not multiple result sets. If a stored

procedure or stored function uses multiple cursor parameters, e.Report Designer Professional and Actuate iServer processes the first cursor parameter and ignores the others. e.Report Designer Professional and Actuate iServer can process a cursor parameter other than the first if the cursor parameter's result columns match the default set in Stored Procedure Data Source Builder. To process a cursor parameter other than the first, override `AcStoredProcedureSource::OpenCursor( )`, as shown in the following example. The cursor parameter name must not include a colon (:).

```
Sub OpenCursor( stmtText As String)
    CursorParameter = "MGRCursor"
    Super::OpenCursor( stmtText )
End Sub
```

Understanding how e.Report Designer Professional works with a stored function

The report design in the following example uses the stored function `REFCUR3.GETMGRDATA` to retrieve data from the `EMP` table in the `SCOTT` database schema. The report user determines which data to retrieve by typing a value for the `DEPTNO` column as an input parameter in Requester.

About the EMP table in the example

The `EMP` table contains the data shown in Figure 11-22.

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	17-DEC-80	800		20
7499	ALLEN	SALESMAN	7698	20-FEB-81	1600	300	30
7521	WARD	SALESMAN	7698	22-FEB-81	1250	500	30
7566	JONES	MANAGER	7839	02-APR-81	2975		20
7654	MARTIN	SALESMAN	7698	28-SEP-81	1250	1400	30
7698	BLAKE	MANAGER	7839	01-MAY-81	2850		30
7782	CLARK	MANAGER	7839	09-JUN-81	2450		10
7788	SCOTT	ANALYST	7566	19-APR-87	3000		20
7839	KING	PRESIDENT		17-NOV-81	5000		10
7844	TURNER	SALESMAN	7698	08-SEP-81	1500	0	30
7876	ADAMS	CLERK	7788	23-MAY-87	1100		20
7900	JAMES	CLERK	7698	03-DEC-81	950		30
7902	FORD	ANALYST	7566	03-DEC-81	3000		20
7934	MILLER	CLERK	7782	23-JAN-82	1300		10

Figure 11-22 The `EMP` table

About the stored function

The following example shows the code for the stored function `REFCUR3.GETMGRDATA`. This stored function defines two cursor parameters, `EMPCursor` and `MGRCursor`. The Stored Procedure Data Source Builder processes the first cursor, `EMPCursor`.

```
create or replace package refCur3 as
    cursor c1 is select ename, deptno from emp;
    cursor c2 is select ename, ename AS manager, hiredate from emp;
```

```

type empCur is ref cursor return c1%ROWTYPE;
type mgrCur is ref cursor return c2%ROWTYPE;

function GetMgrData(indeptno IN NUMBER,EmpCursor in out
empCur,deptAcct OUT NUMBER )
RETURN mgrCur;
END;

create or replace package body refCur3 as
function GetMgrData(indeptno IN NUMBER,EmpCursor in out
empCur,deptAcct OUT NUMBER )
RETURN mgrCur is

    MgrCursor mgrCur;
begin
    open EmpCursor for select ename, deptno from emp where
        deptno = indeptno;
    open MgrCursor for select e.ename, n.ename AS MANAGER,
        e.hiredate
        from emp n, emp e
        where e.mgr= n.empno and e.deptno= indeptno ;
    deptAcct := indeptno + 100;
    return MgrCursor;
end;
end;

```

Figure 11-23 shows the report structure.

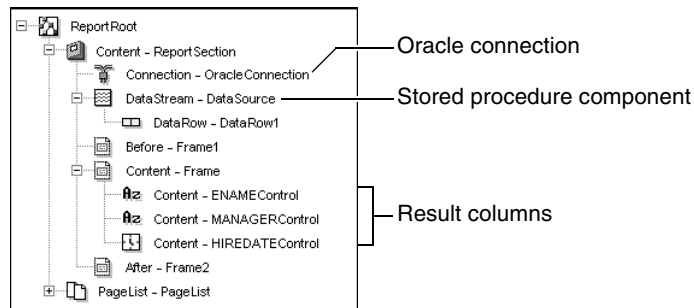


Figure 11-23 The report structure

About the result columns and parameters

Stored Procedure Data Source Builder displays result columns and parameters for the stored function REFCUR3.GETMGRDATA.

Figure 11-24 shows result columns.

Column Name	Actuate Type
ENAME	String
MANAGER	String
HIREDATE	Date

Figure 11-24 The result columns

Figure 11-25 shows the parameters.

Name	Input/Output	Actuate Type
INDEPTNO	IN	Double
EMPCURSOR	IN/OUT	CPointer
DEPTACCT	OUT	Double

Figure 11-25 The parameters

Figure 11-26 shows variables that e.Report Designer Professional creates on the stored procedure component for the DEPTACCT and INDEPTNO parameters.

Variable Name	Type
DEPTACCT_output	Double
FetchLimit	Integer
INDEPTNO	Double
IsAtEnd	Boolean
IsOpen	Boolean
Position	Integer

Figure 11-26 The variables

The Actuate Basic code that is generated for the report design declares a CPointer parameter type for EMPCURSOR and assigns the cursor parameter to an Actuate cursor, AcDBCursor:

```
Class DataSource Subclass Of AcStoredProcedureSource
```

```
Dim DEPTACCT_output As Double
Static INDEPTNO As Double

Sub SetProperties ( )
    Super::SetProperties ( )
    ProcedureName = "REFCUR3.GETMGRDATA"
    OwnerName = ""
    QualifierName = "SCOTT"
    QualificationOption = "UNQUALIFIED"
    ReturnParameter = ":acProcStatus"
    ActualParameters = " :INDEPTNO ,      :EMPCURSOR ,      :DEPTACCT"
    CursorParameter = "acProcStatus"
End Sub

Sub BindStaticParameters( stmt As AcDBStatement )
    stmt.DefineProcedureInputParameter ( "INDEPTNO" ,
        INDEPTNO )
    stmt.DefineProcedureOutputParameter ( "EMPCURSOR" ,
        V_CPOINTER, NULL )
    stmt.DefineProcedureOutputParameter ( "DEPTACCT" ,
        V_DOUBLE )
    stmt. DefineProcedureReturnParameter ( "acProcStatus" ,
        V_CPOINTER )
End Sub

Sub BindDataRow( cursor As AcDBCursor )
    cursor.BindColumn( 1, "NewReportApp::DataRow" , "ENAME" )
    cursor.BindColumn( 3, "NewReportApp::DataRow" ,
        "HIREDATE" )
    cursor.BindColumn( 2, "NewReportApp::DataRow" ,
        "MANAGER" )
End Sub

Sub GetOutputParameters( cursor As AcDBCursor )
    DEPTACCT_output = cursor.GetOutputParameter( "DEPTACCT" )
End Sub
```

Supplying parameter values

When a user runs the report, Requester prompts the user to supply a value for the input parameter, INDEPTNO, as shown in Figure 11-27.

In the EMP table, the DEPTNO column contains the values 10, 20, and 30.

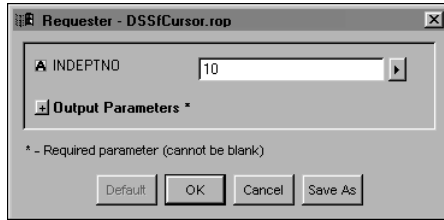


Figure 11-27 Prompting the user to supply a value

Viewing the report results

As shown in Figure 11-28, the report displays the data in the result columns ENAME, MANAGER, and HIREDATE based on the value of INDEPTNO.

CLARK	KING	6/9/81
MILLER	CLARK	1/23/82

Figure 11-28 Report output, displaying the values for department specified in the parameter

Customizing access to handle specific databases or multiple result sets

Typically, you use Stored Procedure Data Source Builder in e.Report Designer Professional to create data sources that use a stored procedure as a source of data. In some cases, however, you override methods in Actuate Basic to customize the way an application accesses the stored procedure and its resulting data. This customization enables you to work with a database that Stored Procedure Data Source Builder does not support, handle multiple cursors that a parameter returns, or meet other special requirements.

Accessing a stored procedure

To call a stored procedure from an Actuate report, you complete the following steps in this order:

- Connect to the database.
- Create and prepare the statement to execute the stored procedure using the connection's `Prepare()` method.
- If you are passing a value or values to the stored procedure, define the procedure input parameters using the statement's `DefineProcedureInputParameter()` method. Do not embed the input parameter definitions in the statement itself.

- To get a value from the stored procedure:
 - Define output parameters using the statement's `DefineProcedureOutputParameter()` method.
 - Call the `StartNextSet()` method.
 - Execute the stored procedure using `Execute()`.
 - Get the output parameter value or values using `GetOutputParameter()`.
- If the stored procedure returns rows:
 - Create a cursor using the statement's `AllocateCursor()` method.
 - Bind columns to data row variables using the cursor's `BindColumn()` method.
 - Create the data-row object using `New()`.
 - Retrieve the rows using the cursor's `Fetch()` method.
- If the stored procedure returns a status, get the return status value using `GetProcedureStatus()`.

Mapping Actuate variable types and Visual Basic type codes

When you use `DefineProcedureOutputParameter()`, you must specify the appropriate Actuate type code. This type code maps to the database data type of the stored-procedure parameter to which the call refers.

Table 11-1 shows the Actuate variable types and the corresponding Visual Basic type code.

Table 11-1 Actuate variable types and the corresponding Visual Basic type code

Actuate variable type	Type code to use
Currency or Variant	V_CURRENCY
Date or Variant	V_DATE
Double or Variant	V_DOUBLE
Integer or Variant	V_INTEGER
Long or Variant	V_LONG
Single or Variant	V_SINGLE
String or Variant	V_STRING

The following statements are examples of how to map the data types in `DefineProcedureOutputParameter()`:

```
statement.DefineProcedureOutputParameter("@charboy", V_STRING)
statement.DefineProcedureOutputParameter("@int_out", V_INTEGER)
```

Working with an Oracle stored procedure

The following example shows the code in Actuate Basic that prepares, passes input parameters to, executes, and obtains output parameters from an Oracle stored procedure that does not return a cursor:

```
Function Start( ) As Boolean
    Dim stmtText As String
    Dim stmt As AcDBStatement
    Dim Conn As AcDBConnection
    Dim arg_in_date As Date, arg_out_date As Date, tempDate As
        Date
    Start = Super::Start( )
    Set Conn = GetConnection( )
    stmtText = "BEGIN sp_date_test(:arg_in_date, :arg_out_date);
        END;"
    Set stmt = conn.Prepare(stmtText)
    If stmt Is Nothing Then
        Exit Function
    End If

    arg_in_date = CDate("01/01/2000")
    If stmt.DefineProcedureInputParameter("arg_in_date",
        arg_in_date) = 0
    Then
        Exit Function
    End If

    If stmt.DefineProcedureOutputParameter("arg_out_date",
        V_DATE) = 0 Then
        Exit Function
    End If

    If stmt.Execute( ) = 0 Then
        stmtText = Conn.GetSpecificErrorText( )
    Exit Function
    End If

    tempDate = stmt.GetOutputParameter("arg_out_date")

End Function
```

Working with a stored procedure's return value

The following example shows the code for a stored procedure and a section of Actuate Basic code that passes input parameters to the stored procedure and extracts the return value. The following stored procedure takes an input

parameter value (a name) and uses that value to determine what value (the associated ID) to return to e.Report Designer Professional or Actuate iServer.

```
-- Create the procedure spfunc
CREATE OR REPLACE FUNCTION spfunc ( p_name IN VARCHAR )
-- Declare output parameter
RETURN NUMBER
AS
    l_id NUMBER;
-- Declare input parameter
BEGIN
    SELECT id INTO l_id FROM sproctable WHERE name = p_name;
    RETURN ( l_id );
END spfunc;
/
```

Working with a stored procedure to return an ID

The following Actuate Basic code executes the stored procedure to determine the ID that is associated with a name. The code passes the input parameter value, the name, to the stored procedure. The stored procedure uses the name to determine its ID and returns the ID.

```
Sub TestSPStatement( connection As AcDBConnection )
    Dim statement As AcDBStatement
    Dim id As Long
    Dim newId As Long
    Dim name As Variant

    ' Prepare the statement
    Set statement = connection.Prepare("BEGIN :p_id := spfunc
(:p_name); END;" )
    If statement Is Nothing Then
        PrintString ( "Failed to prepare statement." )
        PrintString ( connection.GetSpecificErrorText( ) )
        PrintString ( connection.GetGeneralErrorText( ) )
        Exit sub
    End If

    ' Define the input parameter; pass the value "John Smith" to
    the procedure name = "John Smith"
    If statement.DefineProcedureInputParameter ( "p_name",
name ) = 0 Then
        PrintString ( "Failed to define input parameter" )
        PrintString ( connection.GetSpecificErrorText( ) )
        PrintString ( connection.GetGeneralErrorText( ) )
        Exit sub
    End Ifu
```

```

' Define the output parameter
If statement.DefineProcedureOutputParameter ( "p_id",
    V_INTEGER ) = 0
Then
    PrintString ( "Failed to define output parameter" )
    PrintString ( connection.GetSpecificErrorText( ) )
    PrintString ( connection.GetGeneralErrorText( ) )
    Exit sub
End If

' Execute spfunc procedure
If statement.Execute( ) = 0 Then
    PrintString ( "Stored function spfunc execution failed." )
    PrintString ( connection.GetSpecificErrorText( ) )
    PrintString ( connection.GetGeneralErrorText( ) )
Else
    PrintString ( "Stored function spfunc execution success." )
End if

' Get the output parameter value, the id associated with
  "John Smith"
newId = statement.GetOutputParameter ( "p_id" )
Print #1, "Function Return Value - id = ", newId
End Sub

```

Accessing multiple result sets from Oracle stored procedures

This section describes how to access multiple result sets that Oracle stored procedures or stored functions return. This section applies only to using Oracle9i run-time clients.

Stored Procedure Data Source Builder supports only stored procedures that return one result set. To process multiple result sets, you customize Actuate Basic methods. Use the stored procedure's output parameter, declared as a cursor variable of type CPointer. Then, open and fetch data rows from the cursor variable. Each cursor variable provides access to a single result set.

Before you write the custom Actuate Basic code to execute a stored procedure or stored function, you must know the definition of that procedure or function. The definition includes the data type of each parameter and the type of result set that a cursor variable returns. If you are processing multiple result sets, you must call the appropriate Actuate Basic methods for each result set's cursor variable. You also must know the structure and definition of the stored procedure to write custom code to execute it and process its result sets.

A cursor variable can be either strong or weak. A strong cursor specifies a return type with the exact table and columns or row structure to return. A weak cursor supports determining the return type when the report runs.

A stored procedure or stored function returns different columns depending on the values of the input parameters and on how they are defined. If this is the case, the columns in the result set can fail to correspond to the controls in your report design.

Adding support in Actuate Basic methods

First, specify the call to the Oracle stored procedure in the `AcDBStatement::Prepare()` method. For example:

```
stmtText = "BEGIN :mgrCursor := refcur3.GetMgrData (:DEPTNO,
            :empCursor, :deptAcct ); END;"
Set stmt = New AcDBStatement ( GetDBConnection( ) )
If Not stmt.Prepare( stmtText ) Then
...

```

To declare an Oracle stored procedure's output parameter for a result set:

```
AcDBStatement::DefineProcedureOutputParameter
    ("CursorParameterName", V_CPOINTER )

```

where `CursorParameterName` is the name of the output parameter, without the colon (:). Enclose the parameter name in quotation marks.

Assign the output parameter to an Actuate cursor, so you can use `AcDBCursor` methods to bind, open, and fetch data rows. For example, type:

```
AcDBStatement::AllocateCursor ( "CursorParameterName" )

```

where `CursorParameterName` is the name of the cursor parameter, without the colon (:). Enclose the cursor parameter name in quotation marks.

Actuate Basic closes special cursor variables when you close the associated `AcDBCursor`.

Specify the expected return row structure of a cursor variable by calling `AcDBCursor` methods. For example, call:

```
AcDBCursor::BindColumn ( ColumnPositionID, DataRowClassName,
    DataRowVarName )

```

An Oracle stored function returns a value. Use either of the following methods to access the stored function's return value:

- `AcDBStatement::GetProcedureStatus()`
- `AcDBCursor::GetProcedureStatus()`

The return value's data type is any data type that Oracle supports, except `REF_CURSOR`. Supported Oracle data types map to Actuate data types.

If the return value is a `REF_CURSOR`, use:

```
AcDBStatement::AllocateCursor ( "CursorParameterName" )

```

where `CursorParameterName` is the name of the return parameter that is specified in the call to the stored function, without the colon (:).

Using the `CPointer` type with another stored procedure function

You cannot use the cursor variable `CPointer` type with the following methods:

- `AcDBStatement::GetOutputParameter()`
- `AcDBStatement::GetProcedureStatus()`
- `AcDBCursor::GetOutputParameter()`
- `AcDBCursor::GetProcedureStatus()`

The following example displays the names of department employees, the employees' managers, and the employees' hire dates. Users specify the department number to display the data for that department. The example uses the Oracle sample database installed with the Oracle product.

```
-- ref cursor stored function in scott/tiger database
-- where a cursor is returned as a stored function value,
-- in addition to a result set in the output parameter
create or replace package refCur3 as
cursor c1 is select ename, deptno from emp;
cursor c2 is select ename, ename AS manager, hiredate from emp;
type empCur is ref cursor return c1%ROWTYPE;
type mgrCur is ref cursor return c2%ROWTYPE;
function GetMgrData(indeptno IN NUMBER,EmpCursor in out empCur,
                    deptAcct OUT NUMBER )
    RETURN mgrCur;
END;
create or replace package body refCur3 as
    function GetMgrData(indeptno IN NUMBER,EmpCursor in out
        empCur, deptAcct OUT NUMBER )
        RETURN mgrCur is
            MgrCursor mgrCur;
    begin
        open EmpCursor for select ename, deptno from emp
            where deptno = indeptno;
        open MgrCursor for select e.ename, m.ename AS MANAGER,
            e.hiredate
            from emp m, emp e
            where e.mgr= m.empno and e.deptno= indeptno;
        deptAcct := indeptno + 100;
        return MgrCursor;
    end;
end;
```

The example shows the code for an Oracle stored function and a section of the Actuate Basic program that calls it. The Actuate Basic program calls the Oracle stored function with two cursor parameters defined. One of the cursors is the function's return value.

Define the cursors as variables in a subclass of `AcDatabaseSource`. For example:

```
Dim theEmpCursor As AcDBCursor
Dim theMgrCursor As AcDBCursor
```

The example defines two variables in the data stream component:

- `INDEPTNO`, an input parameter to specify the department number that the user enters
- `DEPTACCT_output`, an output parameter that returns the department's account number

Fetching the data row

Override the `Fetch( )` method to fetch rows from the cursor variable:

```
Function Fetch( ) As AcDataRow
'   Set Fetch = Super::Fetch( )
    Set Fetch = myFetch( theMgrCursor )
End Function
```

Calling the Oracle stored procedure with result sets

Add custom Actuate Basic code to call the stored procedure. Add the code in the data stream component of the report design.

The following code example declares input and output parameters, result set cursors, and calls the Oracle stored procedure:

```
Sub OracleSPResults( )

    Dim stmtText As String
    Dim stmt As AcDBStatement
    Dim i_deptId As Integer
    ' Format SQL statement text to call a stored procedure with
      result sets
    stmtText = "BEGIN :mgrCursor := refCur3.GetMgrData( :deptNo,
      :empCursor, :deptAcct ); END;"

    ' Prepare the statement
    Set stmt = New AcDBStatement( GetDBConnection( ) )
    If Not stmt.Prepare( stmtText ) Then
        GetDBConnection( ).RaiseError( )
    Exit Function
End If
```

```

' Define Input and output parameters for stored procedure
i_deptId= 20
stmt.DefineProcedureInputParameter ( "deptNo", i_deptId )

stmt.DefineProcedureOutputParameter( "empCursor", V_CPOINTER,
    NULL )

stmt.DefineProcedureOutputParameter( "deptAcct", V_INTEGER )
stmt.DefineProcedureReturnParameter( "mgrCursor", V_CPOINTER )

' Make optional call to Execute( ) in order to have access to
' output parameter values before calling OpenCursor( )
If Not stmt.Execute( ) Then
    ShowFactoryStatus( "Stored procedure execution failed." )
Exit Function
Else
    ShowFactoryStatus( "Stored procedure execution succeeded." )
End if

' Get the output parameter values to static variable defined
' in OraStoredProcExampleApp
g_deptAcctNo = stmt.GetOutputParameter ( "deptAcct" )

' Now allocate an AcDBCursor for the defined cursor parameter
' and assign to the instance variable
Set theEmpCursor = stmt.AllocateCursor( "empCursor" )

If Not theEmpCursor.OpenCursor( ) Then
    GetDBConnection( ).RaiseError( )
Exit Function
End If

' Bind the first result set columns to the combined data row
theEmpCursor.BindColumn( 1,
    "OraStoredProcExampleApp::DataRow", "ename" )
theEmpCursor.BindColumn( 2,
    "OraStoredProcExampleApp::DataRow", "deptno" )

' Allocate another AcDBCursor for the second result set
Set theMgrCursor = stmt.AllocateCursor( "mgrCursor" )

If Not theMgrCursor.OpenCursor( ) Then
    GetDBConnection( ).RaiseError( )
Exit Function
End If

' Bind the second result set columns to the combined data row
theMgrCursor.BindColumn( 1,
    "OraStoredProcExampleApp::DataRow", "empName" )
theMgrCursor.BindColumn( 2,
    "OraStoredProcExampleApp::DataRow", "mgrName" )

```

```
theMgrCursor.BindColumn( 3,  
    "OraStoredProcExampleApp::DataRow", "hireDate" )  
  
    ' Let the Fetch( ) method takes care of fetching and  
      combining each row of data  
  
End Sub
```


Part Three

**Using Actuate open data access
technology**

Accessing a custom data source

This chapter contains the following topics:

- About accessing additional types of data sources
- Accessing a custom data source using an ODA driver
- Building a custom data stream component
- Providing access to data during report generation
- Providing configuration information about a custom data driver
- Installing a custom data driver

About accessing additional types of data sources

A report developer can access a variety of data sources using predefined data drivers in e.Report Designer Professional. To access other types of data, you can create a custom data driver. e.Report Designer Professional enables this capability by supporting open data access (ODA) drivers.

A report developer chooses a data source from a list of data source types in e.Report Designer Professional. The list does not distinguish between custom and native data sources. When a user chooses a data source type, the driver provides a design-time user interface for querying the data source.

An ODA driver supports both design-time and run-time functionality. e.Report Designer Professional uses the driver to build a connection to the data source, retrieve parameter and data row definitions, and compile these definitions for use at run-time. At run-time, Actuate iServer loads the driver. Then, the driver creates the connection and data source instance and fetches the requested data.

Each ODA driver supports one type of connection and can support multiple instances of that connection type. Each driver can support multiple data sources. The driver must be installed on the system where you design the report and on Actuate iServer where you run the report. For an example of an ODA driver that includes the configuration file and data source builder, see the open data access flat file example in `\Actuate11\oda\Examples\FlatFileExample`. This example also includes run-time classes.

Accessing a custom data source using an ODA driver

To provide an open data access (ODA) driver for a custom data source, perform the following steps:

- Write the code that implements the ODA driver:
 - In e.Report Designer Professional, create a file that contains an ODA connection class and an ODA data stream class. The ODA connection class subclasses `AcOdaConnection`, and the ODA data stream class subclasses `AcOdaSource`. For each class, you must add any custom properties that are necessary to communicate with the ODA driver. As best practice, provide these classes in a report object library (.rol) file. By providing them in an ROL, you can use these classes in multiple reports. You specify the library name and the class names in the open data source's configuration file, `odaconfig.xml`.
 - Create a custom data source builder. This component reads and writes XML files that e.Report Designer Professional uses to define the data source's parameters and properties. e.Report Designer Professional uses

the data source builder of the ODA data driver to support the data source's user interface.

- Create the run-time components of the driver, implementing the ODA run-time Java interfaces. The Factory uses these run-time components to generate a report. For more information about the Java interfaces to use, see "Providing access to data during report generation," later in this chapter.
- Write a configuration file for the driver. The configuration file specifies the ODA interface version of the driver. This file also defines the structure, contents, and semantics of requests and responses between the custom data source and e.Report Designer Professional or Actuate iServer.
- Install the custom data driver and configuration file:
 - Install the custom data driver and configuration file on the same file system as e.Report Designer Professional. When e.Report Designer Professional starts, it caches the custom data driver. Any modifications that you make to the TraceLogging section in odaconfig.xml take effect on the next report generation. For any other changes to the configuration file or the driver to take effect, you must close and reopen e.Report Designer Professional. Closing e.Report Designer Professional releases the cached driver and configuration file. Opening e.Report Designer Professional reloads the more recent version of each.
 - Install the custom data driver and configuration file on the Actuate iServer node that runs the report. If you modify the driver or configuration file after you install them on Actuate iServer, you do not need to restart Actuate iServer to implement your modifications.
- Develop a report that accesses data through the custom data driver. To access the custom data source in a report design, you use a data stream component. You must provide a connection class and one or more data source classes. You access the classes from the report object library (.rol) file that you created in the first step of this process.
- Upload the report executable file to an Encyclopedia volume on the Actuate iServer node that hosts the custom data driver.
- Run the report. You can run a report that uses a custom data source from either e.Report Designer Professional or Actuate iServer. During report generation, a custom data driver performs the following tasks:
 - Opens the data source
 - Accepts values for each data source parameter, if any exist
 - Fetches each data row
 - Closes the data source

Building a custom data stream component

A custom data source developer uses the open data access framework to provide a custom data source builder. An ODA driver's data source builder is a tool that defines the structure of a data stream component. e.Report Designer Professional runs the data source builder executable file when a report developer defines a custom data source. Typically, the data source builder provides a user interface to enable the report developer to refine data source properties. For example, a typical user interface supports the report developer selecting the fields that the data source returns.

The ODA driver's data source builder performs the following tasks:

- Creates a new connection to the custom data source.
- Provides an optional data source builder interface for a user to select and define a data stream.
- Reads the request for data from e.Report Designer Professional.
- Returns parameter and result set definitions for the data stream, the data stream name and type, the run-time command text to execute, and any public or private properties. e.Report Designer Professional maps the result-set definition to the DataRow component and its variables in the report design.

Communicating with a custom data source builder

e.Report Designer Professional communicates with the custom data source builder in a request-and-response format, using XML files. The use of XML facilitates language-independent communication between the data source builder and e.Report Designer Professional. The data source builder must parse the XML request file and populate the XML response file. These two files have a similar structure. The term, data-stream definition file, describes both of these files.

By default, e.Report Designer Professional creates temporary data-stream definition files for the request and response. To retain these files for debugging an ODA driver, create a REG_DWORD registry key, KeepOdaRequestResponseFiles. Create this key in HKEY_CURRENT_USER/Software/Actuate/e.Report Designer Professional <release>/Settings, where <release> is the release number of e.Report Designer Professional. Set the value of the key to 1.

The temporary files each are named odaXXX.tmp, where XXX is a unique, alphanumeric value that Microsoft Windows supplies. e.Report Designer Professional saves the files to the directory that is specified by the TEMP environment variable. Typically, this directory is \Documents and Settings \<your user name>\Local Settings\Temp.

The DesignTimeInterface element in the ODA driver's odaconfig.xml file specifies the data source builder executable file and its command-line arguments.

e.Report Designer Professional passes the names of the data-stream definition files to the data source builder as additional command-line arguments.

About data-stream definition file elements

An XML request file contains a `DesignSessionRequest` element. An XML response file contains a `DesignSessionResponse` element. Both of these elements contain a `DataStreamDefinition` element and a `PublicConnectionProperties` element. The `DataStreamDefinition` element defines characteristics of the data stream, such as the command or query to execute, input and output parameters to use, and public and private properties of the data stream. The `PublicConnectionProperties` element defines the custom values required for connecting to the underlying data source. `DesignSessionResponse` contains a `ResultSetDefinition` element, which defines the metadata of the data fields that the ODA driver returns.

The following sections list and describe some of the principal elements of a data-stream definition file.

DataStreamDefinition element

Properties that define the data stream. `DataStreamDefinition` is an element of both `DesignSessionRequest` and `DesignSessionResponse`. Elements of `DataStreamDefinition` include:

- **Command.** The command or query to execute to retrieve the result set. Command or query syntax is specific to the data source and must be recognized by the driver when the report runs.
- **InputParameters.** Definitions of input parameters to map as report parameters.
- **Name.** The external name of the data stream.
- **OutputParameters.** Definitions of output parameters.
- **PrimaryResultSet.** The `ResultSetDefinition` for the result set that the data stream returns. The result set maps to a `DataRow` component.
- **PrivateProperties.** Properties of the data source. A report developer cannot edit these properties.
- **PublicProperties.** Editable properties of the custom data source.
- **ShortDescription.** A short business description of the data stream.
- **Type.** The type of data source as defined by the ODA driver.

DesignSessionRequest element

The request that defines the data that e.Report Designer Professional passes to an ODA data source builder at the start of the design session. Elements of `DesignSessionRequest` include:

- **DataStreamDefinition.** Properties of the custom data stream component.
- **ExternalDesignState.** An optional element that stores the private state of the data source builder when the builder last closed.
- **PrivateConnectionProperties.** Properties of the connection that a report developer cannot edit.
- **PublicConnectionProperties.** Properties of the connection that a report developer can edit.
- **SessionLocale.** The user locale on the machine that hosts e.Report Designer Professional. SessionLocale can differ from the default locale set in e.Report Designer Professional. If your data source builder does not support all locales, you must handle any unsupported locale.

DesignSessionResponse element

Properties of the design-time interface that a data source builder returns to e.Report Designer Professional at the end of a design session. Elements of DesignSessionResponse include:

- **DataStreamDefinition.** Properties that define the data stream.
- **ExternalDesignState.** An optional element that retrieves the previous private state of the data source builder when the builder closes.
- **PrivateConnectionProperties.** Properties of the connection that a report developer cannot edit.
- **PublicConnectionProperties.** Properties of the connection that a report developer can edit.
- **ResponseState.** The state of the ODA data source builder when it closes.

ResultSetDefinition element

The definition of a single result set that the data stream returns. A data stream must have at least one result set. Each ResultSetDefinition is a named collection of data columns. A result set maps to a DataRow component. Elements of ResultSetDefinition include:

- **Name.** The name of the result set. A value is required if the data source returns multiple result sets.
- **ResultSetColumns.** The collection of data columns in the result set.

Providing access to data during report generation

To retrieve data from a custom data source, create the run-time components of the open data access (ODA) custom data driver. Then, install them on the system that

hosts e.Report Designer Professional and on Actuate iServer. During report generation, the ODA driver performs the following tasks:

- Connects to the open data source
- Handles input and output parameters
- Prepares the statement that requests data from the data source
- Executes the statement
- Manages each result set that the statement returns, passing data rows to the Factory
- Closes the data source

The run-time components of an ODA driver retrieve data using Java interfaces. The interfaces pass data between the custom data source and the Factory. A Java archive file, `AcOdaInterfaces.jar`, contains the class files for the interfaces and class. All interfaces and the class are in the `com.actuate.oda` package. Table 12-1 describes the interfaces and classes for writing an ODA driver.

Table 12-1 ODA driver classes and interfaces

Class or interface	Description
<code>FileHandler</code>	A class that publishes log records to a specified file.
<code>Filter</code>	An interface that provides additional criteria for determining whether to log a record.
<code>Handler</code>	An abstract class that provides classes to support publishing log records.
<code>ICallStatement</code>	An interface for callable statements, which can have output parameters and can return one or more result sets by name. All callable statements, including those for stored procedures and R/3 BAPIs, implement this interface.
<code>IConnection</code>	The connection interface.
<code>IConnectionMetaData</code>	Comprehensive information about the connection.
<code>IDataSourceMetaData</code>	Comprehensive information about the data source.
<code>IParameterMetaData</code>	Metadata for parameters of a prepared statement.
<code>IRResultSet</code>	The interface for all result sets.
<code>IRResultSetMetaData</code>	The metadata interface for a result set.
<code>IRowSet</code>	An interface for representing complex structures or tables.
<code>IStatement</code>	The root interface in the statement hierarchy. Returns one or more result sets in sequence.

(continues)

Table 12-1 ODA driver classes and interfaces (continued)

Class or interface	Description
Level	A class that represents logging levels such as INFO, WARNING, and SEVERE.
LogFormatter	An abstract class that provides classes to convert log records into formatted strings.
LogManager	A class that creates and retrieves Loggers.
LogRecord	A class that contains information that can be logged.
Logger	A class represents the log for an application.
LoggingErrorHandler	A class that can be associated with a Handler to process any exceptions that occur during logging.
OdaException	A class that returns information about a data source access error or other errors.
SimpleFormatter	A class that extends LogFormatter and formats a log record.
SortSpec	A class that represents the sorting specification for the result set or sets of an IStatement.
StreamHandler	A class that extends Handler to support logging with output streams.
StringSubstitutionUtil	A class that finds and replaces strings.

The Factory instantiates the class that is defined by the DriverInitEntryPoint element of the RunTimeInterface element of odaconfig.xml. This class must implement the IConnection interface. Methods that are defined by this interface provide access to parameters, statements, and result sets.

e.Report Designer Professional accesses the driver's odaconfig.xml file to obtain the logging configuration data. It then passes this information to the ODA run-time driver for processing. If the logging configuration information in odaconfig.xml is updated without shutting down the host process, e.Report Designer Professional detects the changes and passes the updated log configuration using IConnectionFactory.setLogConfiguration(). The ODA driver can then use LogManager.createLogger() to set up the appropriate logger.

Providing configuration information about a custom data driver

The configuration file, odaconfig.xml, specifies the structure, contents, and semantics of requests and responses between the open data source and e.Report Designer Professional or Actuate iServer. For example, odaconfig.xml defines the connection type and data sources that the driver supports and maps external data

types to ODA data types, which e.Report Designer Professional subsequently maps to its data types. You also specify the ODA interface version of the driver.

e.Report Designer Professional reads this configuration file when it starts. Actuate iServer uses the configuration file to load the custom data driver during report generation. Each ODA driver has a unique configuration file, `odaconfig.xml`. When e.Report Designer Professional starts, it gathers information about the driver by reading `odaconfig.xml`. During report generation, the Factory uses the configuration file to load the ODA driver.

When you write an ODA configuration file, the following rules apply:

- Name the file `odaconfig.xml`.
- Install `odaconfig.xml` in a specific location on the machine that hosts e.Report Designer Professional and on Actuate iServer.
- Follow the XML schema specified for the ODA configuration file.

Installing a custom data driver

To use a custom data driver in both the design-time and run-time environments, install the ODA driver on a machine that hosts e.Report Designer Professional and on the Actuate iServer node that runs the report. Install the driver in a folder named `oda` in the directory tree of the Actuate release. You must adhere to the expected directory structure. Each driver must reside in a subdirectory of the `\oda` folder. Do not add levels to the structure. Do not group multiple drivers in a single subdirectory. If you use multiple versions of a driver, create a subdirectory for each version in `\oda` and indicate the version number or other identifier in the file name, as shown in the following examples:

```
\Program Files\Actuate11\oda\CustomDriver_version1.0
\Program Files\Actuate11\oda\CustomDriver_version1.1
```

The name that you assign to the driver must be unique. The report executable file passes the name to the Factory. The Factory loads the driver by looking up the unique name. The driver name is case-sensitive and must not contain characters that the operating system does not recognize.

The name of the driver subdirectory must be exactly the same as the value of the `DriverName` element of `odaconfig.xml`. For example, create a folder named `CustomDriver_version1.0` for the driver described in the following XML code:

```
<?xml version="1.0" encoding="UTF-8"?>
  <OpenDataAccessConfig>
    <DriverName>CustomDriver_version1.0</DriverName>
    ...
  </OpenDataAccessConfig>
```

How to install an ODA driver

To use the ODA driver on an Actuate iServer, install the ODA driver on all the nodes that run the report. To use the ODA driver in e.Report Designer Professional, install the driver on the same machine as e.Report Designer Professional.

- 1 Create or navigate to a directory named `\oda` in the home directory for the Actuate release. For example, on a Windows XP platform that runs Actuate 11 create the new directory using the following path:

```
\Program Files\Actuate11\oda
```

To install the ODA driver on an Actuate iServer node that runs on a UNIX system, use the following path:

```
$AC_SERVER_HOME/oda
```

- 2 In `\oda`, create a separate subdirectory for each ODA driver, as shown in the following example:

```
\Program Files\Actuate11\oda\XMLDataDriver
```

On a UNIX platform, the path is:

```
$AC_SERVER_HOME/oda/CustomDriver
```

- 3 Place the following files in the subfolder:
 - The ODA driver configuration file, `odaconfig.xml`.
 - Any report library files that are required.
 - For an Actuate iServer node, you must also place in the subdirectory any Java driver files that the run-time interfaces require.
 - For a system that hosts e.Report Designer Professional, you also must place the following files in the subfolder:
 - The driver design-time executable file, if there is one
 - The Java driver files that the design-time and run-time interfaces require

Configuration file XML schema reference

This chapter contains the following topics:

- About the configuration file for a custom data source
- Understanding the schema of `odaconfig.xml`
- Using the elements of `odaconfig.xml`

About the configuration file for a custom data source

A report developer chooses a data source from a list of data source types in e.Report Designer Professional. A configuration file named `odaconfig.xml` provides e.Report Designer Professional with the information required to access a custom data source. To specify the name that appears in the list of data source types, set the connection's `DisplayName` property in `odaconfig.xml`. When the user chooses the data source type, the driver provides a design-time user interface to specify the query.

Each custom data source requires a unique configuration file. You install the file on the same file system as e.Report Designer Professional. When e.Report Designer Professional starts, it reads the configuration file to gather information about the driver. Actuate iServer uses the configuration file to load the custom data driver during report generation.

The configuration file specifies the ODA interface version of the driver. This file also defines the structure, contents, and semantics of requests and responses between the open data source and e.Report Designer Professional or Actuate iServer. For example, `odaconfig.xml` defines the connection type and data sources that the driver supports and maps external data types to ODA data types, which e.Report Designer Professional maps to its data types. When e.Report Designer Professional starts, it reads `odaconfig.xml`.

For an example of an ODA driver including the configuration file, data source builder, and run-time classes, see the flat file data source in `\Actuate11\oda\Examples\FlatFileExample`.

Understanding the schema of `odaconfig.xml`

The following schema shows the structure of the `odaconfig.xml` file. A report developer can modify the values in `odaconfig.xml` for a particular driver. The XML parser that Actuate uses does not recognize changes to the schema itself. For a definition of each element, see "Using the elements of `odaconfig.xml`," later in this chapter.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--W3C Schema generated by XMLSPY v5 rel. 4 U
(http://www.xmlspy.com)-->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified">
  <xs:element name="AlternativeOdaDataTypes">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="OdaScalarDataType"
          maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

```

        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="Connection">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="Name" type="xs:string"/>
            <xs:element name="DisplayName" type="xs:string"/>
            <xs:element ref="Properties" minOccurs="0"/>
            <xs:element ref="DesignerSpecificProperties"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="DataSource">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="Name" type="xs:string"/>
            <xs:element name="DisplayName" type="xs:string"/>
            <xs:element ref="Properties" minOccurs="0"/>
            <xs:element name="DataTypeMappings">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element ref="DataTypeMapping"
                            minOccurs="0" maxOccurs="unbounded"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
            <xs:element ref="DesignerSpecificProperties"
                minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="DataSources">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="DataSource" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="DesignTimeInterface">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="Executable" type="xs:string"/>
            <xs:element name="Arguments" type="xs:string"
                minOccurs="0"/>
            <xs:element name="WarnValueFormatAdjustment"
                type="xs:boolean" minOccurs="0"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>

```

```

        </xs:element>
    </xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="DesignerSpecificProperties">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="DesignerSpecific"
                maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="DesignerSpecific"
    type="DesignerSpecificType"/>
<xs:element name="InterfaceType">
    <xs:simpleType>
        <xs:restriction base="xs:string">
            <xs:enumeration value="C"/>
            <xs:enumeration value="Java"/>
        </xs:restriction>
    </xs:simpleType>
</xs:element>
<xs:element name="DataTypeMapping" type="NativeDataTypeType"/>
<xs:element name="OSType">
    <xs:simpleType>
        <xs:restriction base="xs:string">
            <xs:enumeration value="Windows"/>
            <xs:enumeration value="Solaris"/>
            <xs:enumeration value="HP-UX"/>
            <xs:enumeration value="AIX"/>
            <xs:enumeration value="LINUX"/>
            <xs:enumeration value="All"/>
        </xs:restriction>
    </xs:simpleType>
</xs:element>
<xs:element name="OpenDataAccessConfig">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="DriverName" type="xs:string"/>
            <xs:element ref="VendorInfo"/>
            <xs:element name="ODAInterfaceVersion" type="xs:float"/>
            <xs:element ref="DesignTimeInterface"/>
            <xs:element ref="RunTimeInterface"/>
            <xs:element ref="Connection"/>
            <xs:element ref="DataSources"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>

```



```

        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="Properties">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="Property" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="Property" type="PropertyType"/>
<xs:element name="RunTimeInterface">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="InterfaceType"/>
            <xs:element name="DriverInitEntryPoint"
                type="xs:string"/>
            <xs:element ref="DriverLibraries"/>
            <xs:element name="TraceLogging" type="TraceLogging"
                minOccurs="0"/>
            <xs:element name="StreamedResultSetBufferSize"
                type="xs:unsignedShort" minOccurs="0"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="VendorInfo">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="ProductName" type="xs:string"/>
            <xs:element name="Version" type="xs:string"/>
            <xs:element name="Vendor" type="xs:string"/>
            <xs:element name="VendorURL" type="xs:string"
                minOccurs="0"/>
            <xs:element name="VendorPhone" type="xs:string"
                minOccurs="0"/>
            <xs:element name="CopyrightNotice" type="xs:string"
                minOccurs="0"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:complexType name="PropertyType">
    <xs:sequence>
        <xs:element name="Name" type="xs:string"/>
        <xs:element name="DisplayName" type="xs:string"
            minOccurs="0"/>
        <xs:element name="Value" type="xs:string"/>
    </xs:sequence>
</xs:complexType>

```

```

        <xs:element name="SelectValueList" type="ArrayOfString"
            minOccurs="0"/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="NativeDataTypeType">
    <xs:sequence>
        <xs:element name="NativeDataType" type="xs:string"/>
        <xs:element name="NativeDataTypeCode" type="xs:short"/>
        <xs:element ref="OdaScalarDataType"/>
        <xs:element ref="AlternativeOdaDataTypes"
            minOccurs="0"/>
    </xs:sequence>
</xs:complexType>
<xs:element name="OdaScalarDataType">
    <xs:simpleType>
        <xs:restriction base="xs:string">
            <xs:enumeration value="Date"/>
            <xs:enumeration value="Double"/>
            <xs:enumeration value="Integer"/>
            <xs:enumeration value="String"/>
            <xs:enumeration value="Time"/>
            <xs:enumeration value="Timestamp"/>
            <xs:enumeration value="Decimal"/>
            <xs:enumeration value="Blob"/>
            <xs:enumeration value="Clob"/>
        </xs:restriction>
    </xs:simpleType>
</xs:element>
<xs:element name="DriverLibraries">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="LibrariesForOS" maxOccurs="unbounded"/>
            <xs:element name="SetJavaThreadContextClassLoader"
                type="xs:boolean" minOccurs="0"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="LibrariesForOS" type="LibrariesForOSType"/>
<xs:complexType name="ArrayOfString">
    <xs:sequence>
        <xs:element name="String" type="xs:string"
            maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="LibrariesForOSType">

```

```

<xs:sequence>
  <xs:element ref="OSType"/>
  <xs:element name="LibraryName" type="xs:string"
    maxOccurs="unbounded"/>
  <xs:element name="Location" type="xs:string"
    minOccurs="0"/>
</xs:sequence>
</xs:complexType>
<xs:complexType name="DesignerSpecificType">
  <xs:sequence>
    <xs:element name="DesignerName" type="xs:string"/>
    <xs:element name="ListOfProperties">
      <xs:complexType>
        <xs:sequence>
          <xs:element ref="Property" maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="TraceLogging">
  <xs:sequence>
    <xs:element name="LogLevel" type="xs:short"/>
    <xs:element name="LogFilenamePrefix" type="xs:string"/>
    <xs:element name="LogDirectory" type="xs:string"
      minOccurs="0"/>
    <xs:element name="JavaLogFormatterClass"
      type="xs:string" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
</xs:schema>

```

Using the elements of odaconfig.xml

The following example shows a configuration file for accessing a flat file data source. You can view this file in \Actuate11\oda\Examples\FlatFileExample. This example uses a report object library (.rol) file to enable e.Report Designer Professional to access subclasses of AcOdaConnection and AcOdaSource classes.

```

<?xml version="1.0" encoding="UTF-8" ?>
<!-- Configuration file for flat file example of Open Data Access
  Framework -->
<OpenDataAccessConfig
  xmlns:xsd="http://www.actuate.com/XMLSchemas/odaConfigv1">
  <driverName>AcOdaFlatFileExampleDriver</driverName>

```

```

<VendorInfo>
  <ProductName>
    Actuate Open Data Access Flat File Example Driver
  </ProductName>
  <Version>8</Version>
  <Vendor>Actuate Corporation</Vendor>
  <VendorURL>http://www.actuate.com</VendorURL>
  <VendorPhone>888-422-8828</VendorPhone>
  <CopyrightNotice>Copyright (C) 2003-2004 Actuate Corporation
  </CopyrightNotice>
</VendorInfo>
<OdaInterfaceVersion>1.2</OdaInterfaceVersion>
<DesignTimeInterface>
  <Executable>javaw</Executable>
  <Arguments>
    -classpath
    c:\oda\AcOdaFlatFileExampleDriver\
      AcOdaFlatFileExampleDesigner.jar;
    c:\oda\AcOdaFlatFileExampleDriver\AcOdaInterfaces.jar;
    c:\oda\AcOdaFlatFileExampleDriver\
      AcOdaFlatFileExampleRuntime.jar;
    c:\oda\AcOdaFlatFileExampleDriver\PullParser_SMALL.jar
    com.actuate.oda.sample.FlatFileExample
  </Arguments>
</DesignTimeInterface>
<RunTimeInterface>
  <InterfaceType>Java</InterfaceType>
  <DriverInitEntryPoint>
    com.actuate.oda.sample.runtime.ConnectionFactoryImpl
  </DriverInitEntryPoint>
  <DriverLibraries>
    <LibrariesForOS>
      <OSType>Windows</OSType>
      <LibraryName>AcOdaFlatFileExampleRuntime.jar
      </LibraryName>
      <LibraryName>AcOdaInterfaces.jar</LibraryName>
    </LibrariesForOS>
  </DriverLibraries>
</RunTimeInterface>
<Connection>
  <Name>OdaFlatFileExampleConnection</Name>
  <DisplayName>Oda Flat file example connection</DisplayName>
  <Properties>
    <Property>
      <Name>SourceFile</Name>
      <Value />
    </Property>
  </Properties>
</Connection>

```

```

</Properties>
<DesignerSpecificProperties>
  <DesignerSpecific>
    <DesignerName>ERD</DesignerName>
    <ListOfProperties>
      <Property>
        <Name>ClassName</Name>
        <Value>OdaFlatFileExampleConnection</Value>
      </Property>
      <Property>
        <Name>DesignLibrary</Name>
        <Value>AcOdaFlatFileExample.rol</Value>
      </Property>
    </ListOfProperties>
  </DesignerSpecific>
</DesignerSpecificProperties>
</Connection>
<DataSources>
  <DataSource>
    <Name>AcODAFlatFileDataSource</Name>
    <DisplayName>Flat file data source example</DisplayName>
    <DataTypeMappings>
      <DataTypeMapping>
        <NativeDataType>String</NativeDataType>
        <NativeDataTypeCode>1</NativeDataTypeCode>
        <OdaScalarDataType>String</OdaScalarDataType>
      </DataTypeMapping>
    </DataTypeMappings>
    <DesignerSpecificProperties>
      <DesignerSpecific>
        <DesignerName>ERD</DesignerName>
        <ListOfProperties>
          <Property>
            <Name>ClassName</Name>
            <Value>ODAFlatFileExampleSource</Value>
          </Property>
          <Property>
            <Name>DesignLibrary</Name>
            <Value>AcOdaFlatFileExample.rol</Value>
          </Property>
        </ListOfProperties>
      </DesignerSpecific>
    </DesignerSpecificProperties>
  </DataSource>
</DataSources>
</OpenDataAccessConfig>

```

An odaconfig.xml configuration file describes an open data source to e.Report Designer Professional and Actuate iServer System. The rest of this section serves as a reference guide for the elements of an odaconfig.xml configuration file.

AlternativeOaDataTypes

A sequence of OdaScalarDataType elements

Schema

```
<xs:element name="AlternativeOaDataTypes">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="OdaScalarDataType"
        maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Elements **OdaScalarDataType**
The ODA data type

ArrayOfString

List of String elements

Schema

```
<xs:complexType name="ArrayOfString">
  <xs:sequence>
    <xs:element name="String" type="xs:string"
      maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

Elements **String**
The String data type

Connection

Each driver supports a single type of connection and can support multiple instances of that connection type.

Schema

```
<xs:element name="Connection">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Name" type="xs:string"/>
      <xs:element name="DisplayName" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

```

        <xs:element ref="Properties" minOccurs="0"/>
        <xs:element ref="DesignerSpecificProperties"/>
    </xs:sequence>
</xs:complexType>
</xs:element>

```

Elements **DesignerSpecificProperties**

Defines a list of custom properties for the ODA connection. The design tool identified by DesignerName in DesignerSpecificType displays these properties.

DisplayName

The connection name that appears in a user interface.

Name

An identifier for the connection.

Properties

An optional element that defines public properties of the connection. A property is a name-value pair. The name is unique. The value is the default value to use in the report design. If specified in the configuration file and supported by e.Report Designer Professional, the value of DisplayName appears in the property editor and SelectList provides a static list of values to select for a property.

DataSource

Each driver can support multiple data sources. A separate DataSource section of the configuration file defines each data source.

Schema

```

<xs:element name="DataSource">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Name" type="xs:string"/>
      <xs:element name="DisplayName" type="xs:string"/>
      <xs:element ref="Properties" minOccurs="0"/>
      <xs:element name="DataTypeMappings">
        <xs:complexType>
          <xs:sequence>
            <xs:element ref="DataTypeMapping"
              minOccurs="1" maxOccurs="unbounded"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element ref="DesignerSpecificProperties"
        minOccurs="1" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

```

Elements **DataTypeMappings**

A sequence of elements that map the data source's native data types to ODA data types. e.Report Designer Professional maps ODA data types to Actuate Basic data types.

DesignerSpecificProperties

Defines a list of custom properties for the ODA data source class. The Actuate design tool identified by DesignerName in DesignerSpecificType displays these properties. DesignerName for e.Report Designer Professional is ERD. Each data source requires a ClassName and a DesignLibrary value.

DisplayName

The name that e.Report Designer Professional displays for the data source.

Name

A unique identifier for the data source.

Properties

An optional element for the public properties of the data source, including the name, value, and optionally other information about the properties. If specified in the configuration file and supported by e.Report Designer Professional, the value of DisplayName appears in the property editor and SelectValueList provides a static list of values to select for a property.

DataSources

A sequence of DataSource elements

Schema `<xs:element name="DataSources">
 <xs:complexType>
 <xs:sequence>
 <xs:element ref="DataSource" maxOccurs="unbounded"/>
 </xs:sequence>
 </xs:complexType>
</xs:element>`

Elements **DataSource**
Defines a data source

DataTypeMapping

An element that maps the data source's native data types to ODA data types. e.Spreadsheet Designer uses the ODA data types. e.Report Designer Professional maps ODA data types to Actuate Basic data types. DataTypeMapping also supports designating optional alternative data types that the user can change in the design environment. Implement all feasible data type mappings to ensure

accurate conversions. For example, if a result set column is a native Integer type, the driver should support getString() to convert the Integer value to a string using a format specific to the data source.

Schema `<xs:element name="DataTypeMapping" type="NativeDataTypeType"/>`

DataTypeMappings

A sequence of DataTypeMapping elements that map the data source's native data types to ODA data types. e.Report Designer Professional maps ODA data types to Actuate Basic data types.

Schema `<xs:element name="DataTypeMappings">
 <xs:complexType>
 <xs:sequence>
 <xs:element ref="DataTypeMapping" maxOccurs="unbounded"/>
 </xs:sequence>
 </xs:complexType>
</xs:element>`

Elements **DataTypeMapping**
Maps a native data type of the data source to an ODA data type.

DesignerSpecific

Defines a list of custom properties for the ODA connection

Schema `<xs:element name="DesignerSpecific" type="DesignerSpecificType"/>`

DesignerSpecificProperties

A sequence of elements that lists custom properties for the ODA connection

Schema `<xs:element name="DesignerSpecificProperties">
 <xs:complexType>
 <xs:sequence>
 <xs:element ref="DesignerSpecific" maxOccurs="unbounded"/>
 </xs:sequence>
 </xs:complexType>
</xs:element>`

Elements **DesignerSpecific**
Lists a custom property for the ODA connection

DesignerSpecificType

Defines a list of custom properties for the ODA connection. The Actuate design tool identified by DesignerName in DesignerSpecificType displays these properties:

- **AddToDriver**
If AddToDriver is false, the Actuate design tool identified by DesignerName does not display this data source in its list of data source types. By default, AddToDriver is true.
- **ClassName**
ClassName, required by e.Report Designer Professional, must specify a subclass of AcOdaConnection.
- **DesignLibrary**
The value of DesignLibrary, required by e.Report Designer Professional, which can be a full path or a file name with a relative path. DesignLibrary can be an ROL, ROD, or BAS file.

Schema

```
<xs:complexType name="DesignerSpecificType">
  <xs:sequence>
    <xs:element name="DesignerName" type="xs:string"/>
    <xs:element name="ListOfProperties">
      <xs:complexType>
        <xs:sequence>
          <xs:element ref="Property" maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
```

Elements **DesignerName**
The Actuate design tool. DesignerName for e.Report Designer Professional is ERD.

ListOfProperties
An unbounded sequence of properties

DesignTimeInterface

Specifies the executable file with which to build the data source

Schema

```
<xs:element name="DesignTimeInterface">
  <xs:complexType>
```

```

<xs:sequence>
  <xs:element name="Executable" type="xs:string"/>
  <xs:element name="Arguments" type="xs:string"
    minOccurs="0"/>
  <xs:element name="WarnValueFormatAdjustment"
    type="xs:boolean" minOccurs="0">
  </xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>

```

Elements **Arguments**

If you use arguments, place them in a single line. In the optional Arguments element, use quotation marks to enclose a command or argument that contains spaces. If you omit the quotation marks, a command error occurs. The following example shows a Java CLASSPATH and Java class:

```

-classpath c:\oda\AcOdaFlatFileExampleDriver
  \AcOdaFlatFileExampleDesigner.jar;
c:\oda\AcOdaFlatFileExampleDriver\AcOdaInterfaces.jar;
c:\oda\AcOdaFlatFileExampleDriver
  \AcOdaFlatFileExampleRuntime.jar;
c:\oda\AcOdaFlatFileExampleDriver\PullParser_SMALL.jar
com.actuate.oda.sample.FlatFileExample

```

Executable

A required element that specifies either an absolute path to an executable file or no path. If you do not specify a path, the executable file must be in either the directory that contains the driver's configuration file or in any directory that the PATH variable contains. In the Executable element, use quotation marks to enclose a command or argument that contains spaces. If you omit the quotation marks, a command error occurs.

WarnValueFormatAdjustment

An optional element that specifies warning the user in the following circumstances. By default, e.Report Designer Professional issues a warning when adjusting a value's format. For example, e.Report Designer Professional can adjust the value 123.45 to 123,45 to convert an input parameter value to the French locale. As another example, e.Report Designer Professional truncates trailing zeroes after a decimal separator in double values. A warning is useful in these cases so that the user can see the change. In other cases, the value format changes are internal and a warning is inappropriate. If you do not want a warning, set the value to False.

DriverLibraries

A sequence of LibrariesForOS elements, optionally paired with SetJavaThreadContextClassLoader elements

Schema `<xs:element name="DriverLibraries">
 <xs:complexType>
 <xs:sequence>
 <xs:element ref="LibrariesForOS" maxOccurs="unbounded"/>
 <xs:element name="SetJavaThreadContextClassLoader"
 type="xs:boolean" minOccurs="0">
 </xs:element>
 </xs:sequence>
 </xs:complexType>
</xs:element>`

Elements **LibrariesForOS**
 Required library files for each supported operating system that hosts the Factory.

SetJavaThreadContextClassLoader

If you create an ODA driver using a Java package such as Log4J that relies on the current thread's context class loader to load related classes, set this optional element to true to avoid class loader conflict. This setting designates the current thread's context class loader to load the driver's run-time libraries. The default value is false.

InterfaceType

Specifies whether the driver is written in C or Java

Schema `xs:element name="InterfaceType">
 <xs:simpleType>
 <xs:restriction base="xs:string">
 <xs:enumeration value="C"/>
 <xs:enumeration value="Java"/>
 </xs:restriction>
 </xs:simpleType>
</xs:element>`

Elements **C**
 Specifies the C language

Java
 Specifies the Java language

LibrariesForOs

A child element of DriverLibraries that defines the required library files for each operating system of a machine that hosts the Factory.

Schema `<xs:element name="LibrariesForOS" type="LibrariesForOSType"/>`

LibrariesForOsType

LibrariesForOsType defines the required library files for each operating system of a machine that hosts the Factory. The following properties define LibrariesForOS:

Schema

```
<xs:complexType name="LibrariesForOSType">
  <xs:sequence>
    <xs:element ref="OSType"/>
    <xs:element name="LibraryName" type="xs:string"
      maxOccurs="unbounded"/>
    <xs:element name="Location" type="xs:string" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
```

Elements **LibraryName**

The name of a driver library, such as a Java archive (.jar) file or an operating system library such as a DLL file on Windows. LibraryName must be a file name and file extension without a path, such as OdaLib.jar. A run-time ODA driver can have multiple libraries. Place all libraries for a driver in the same directory.

Location

An optional element that specifies a path to the directory that contains the driver libraries. If you do not specify a location, the Factory loads the driver libraries from the directory containing the driver.

OSType

The operating system for the specified libraries. Valid values are Windows, Solaris, HP-UX, AIX, Linux, or All.

ListOfProperties

An unbounded sequence of properties

Schema

```
<xs:element name="ListOfProperties">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="Property" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Elements **Property**

Contains the name, value, and optionally other information for a property

NativeDataTypeType

An element that maps the data source's native data types to ODA data types. e.Spreadsheet Designer uses the ODA data types. e.Report Designer Professional maps ODA data types to Actuate Basic data types.

NativeDataTypeType also supports designating optional alternative data types that the report developer can set in the design environment. Implement all feasible data type mappings to ensure accurate conversions. For example, if a result set column is a native Integer type, the driver should support getString() to convert the Integer value to a string using a format specific to the data source.

Schema

```
<xs:complexType name="NativeDataTypeType">
  <xs:sequence>
    <xs:element name="NativeDataType" type="xs:string"/>
    <xs:element name="NativeDataTypeCode" type="xs:short"/>
    <xs:element ref="OdaScalarDataType"/>
    <xs:element ref="AlternativeOdaDataTypes" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
```

Elements **AlternativeOdaDataTypes**

Optional additional data types that e.Report Designer Professional supports for this native type. If an alternative data type is assigned, the report developer can change the data type mapping for a column.

NativeDataType

The data type of the column in the open data source

NativeDataTypeCode

The code the open data source uses for the native data type

OdaScalarDataType

The corresponding ODA data type that e.Report Designer Professional supports

OdaScalarDataType

The types of scalar data types used by ODA

Schema

```
<xs:element name="OdaScalarDataType">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="Date"/>
      <xs:enumeration value="Double"/>
      <xs:enumeration value="Integer"/>
      <xs:enumeration value="String"/>
      <xs:enumeration value="Time"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

```

        <xs:enumeration value="Timestamp"/>
        <xs:enumeration value="Decimal"/>
        <xs:enumeration value="Blob"/>
        <xs:enumeration value="Clob"/>
    </xs:restriction>
</xs:simpleType>
</xs:element>

```

Elements

Blob

Supports a binary large object

Clob

Supports a character large object

Date

Supports a date value

Decimal

Supports a decimal value

Double

Supports a double value

Integer

Supports an integer value

String

Supports a text string

Time

Supports a time value

Timestamp

Supports a timestamp value that contains both date and time information

OpenDataAccessConfig

Specifies the access information for the custom ODA driver

Schema

```

<xs:element name="OpenDataAccessConfig">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="DriverName" type="xs:string"/>
      <xs:element ref="VendorInfo"/>
      <xs:element name="ODAInterfaceVersion" type="xs:float"/>
      <xs:element ref="DesignTimeInterface"/>
      <xs:element ref="RunTimeInterface"/>
      <xs:element ref="Connection"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

```

```

        <xs:element ref="DataSources"/>
    </xs:sequence>
</xs:complexType>
</xs:element>

```

Elements

Connection

Each driver supports a single type of connection and can support multiple instances of that connection type.

DataSources

Each driver can support multiple data sources. A sequence of DataSource elements defines the data sources.

DesignTimeInterface

Specifies the executable file with which to build the data source

DriverName

A unique, case-sensitive identifier for each driver. The name must be the same as the name of the folder containing the ODA driver files. Thus, the name must contain only characters that the operating system can interpret as a part of a folder name.

ODAInterfaceVersion

A mandatory element that supports backward compatibility for future versions of the design-time and run-time interfaces. The value must be a Float. To compile using Actuate 11, the value is 1.3.

RuntimeInterface

Defines the run-time interface.

VendorInfo

Specifies information about the ODA driver and its vendor. e.Report Designer Professional does not use this information.

OSType

The type of operating system that the driver supports

Schema

```

<xs:element name="OSType">
    <xs:simpleType>
        <xs:restriction base="xs:string">
            <xs:enumeration value="Windows"/>
            <xs:enumeration value="Solaris"/>
            <xs:enumeration value="HP-UX"/>
            <xs:enumeration value="AIX"/>
            <xs:enumeration value="LINUX"/>
            <xs:enumeration value="All"/>
        </xs:restriction>
    </xs:simpleType>
</xs:element>

```



```

        </xs:restriction>
    </xs:simpleType>
</xs:element>

```

Elements	AIX
	Supported on IBM AIX operating systems
	All
	Supported on all operating systems
	HP-UX
	Supported on HP-UX operating systems
	LINUX
	Supported on LINUX operating systems
	Solaris
	Supported on Solaris operating systems
	Windows
	Supported on Microsoft Windows operating systems

Properties

An unbounded sequence of properties

Schema	<pre> <xs:element name="Properties"> <xs:complexType> <xs:sequence> <xs:element ref="Property" maxOccurs="unbounded" /> </xs:sequence> </xs:complexType> </xs:element> </pre>
Elements	Property
	Contains the name, value and optionally other information for a property

Property

An element of type PropertyType, containing the name, value, and optionally other information for a property

Schema	<pre> <xs:element name="Property" type="PropertyType"/> </pre>
---------------	--

PropertyType

Contains the name, value, and optionally other information for a property

Schema `<xs:complexType name="PropertyType">`
 `<xs:sequence>`
 `<xs:element name="Name" type="xs:string"/>`
 `<xs:element name="DisplayName" type="xs:string"`
 `minOccurs="0"/>`
 `<xs:element name="Value" type="xs:string"/>`
 `<xs:element name="SelectValueList" type="ArrayOfString"`
 `minOccurs="0"/>`
 `</xs:sequence>`
`</xs:complexType>`

Elements **DisplayName**
An optional string to use when displaying the property to report developers

Name
The unique name for the property

SelectValueList
An optional array of strings to present to the report developer as choices for the property value

Value
The default value to use in the report design.

RunTimeInterface

Defines the run-time interface

Schema `<xs:element name="RunTimeInterface">`
 `<xs:complexType>`
 `<xs:sequence>`
 `<xs:element ref="InterfaceType"/>`
 `<xs:element name="DriverInitEntryPoint" type="xs:string"/>`
 `<xs:element ref="DriverLibraries"/>`
 `<xs:element name="TraceLogging" type="TraceLogging"`
 `minOccurs="0"/>`
 `<xs:element name="StreamedResultSetBufferSize"`
 `type="xs:unsignedShort" minOccurs="0">`
 `</xs:element>`
 `</xs:sequence>`
 `</xs:complexType>`
`</xs:element>`

Elements DriverInitEntryPoint

The entry point for the Factory after it loads the required ODA library. For a Java interface, the value is a custom class that implements IConnectionFactory. The Factory uses this name to create a Java object that creates the Java connection object.

DriverLibraries

Defines any libraries the custom driver uses but does not load. Use this element to list libraries that the Factory should load with the ODA driver. For ODA drivers with InterfaceType Java, these libraries must be JAR or ZIP files. You can define libraries for multiple operating systems using this element. You must include AcOdaInterfaces.jar, which contains the ODA run-time interfaces. For more information about these interfaces, see Chapter 15, “Custom data driver Java reference.”

InterfaceType

The programming language used to write the custom ODA data driver. In Actuate 11, the value must be Java.

LibrariesForOS

Defines the required library files for each supported operating system that hosts the Factory.

StreamedResultSetBufferSize

Optional element for internal use only.

TraceLogging

Optional element that defines the log configuration information. This information is used by the classes and interface in the com.actuate.oda.util.logging package, such as Logger and LogManager. For more information about these classes and interface, see Chapter 15, “Custom data driver Java reference.”

TraceLogging

Defines the log configuration information. This information is used by the classes and interface in the com.actuate.oda.util.logging package, such as Logger and LogManager. For more information about these classes and interface, see Chapter 15, “Custom data driver Java reference.”

Schema

```
<xs:complexType name="TraceLogging">
  <xs:sequence>
    <xs:element name="LogLevel" type="xs:short"/>
    <xs:element name="LogFilenamePrefix" type="xs:string"/>
    <xs:element name="LogDirectory" type="xs:string"
      minOccurs="0"/>
    <xs:element name="JavaLogFormatterClass" type="xs:string"
      minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
```

```

    </xs:sequence>
</xs:complexType>

```

Elements **JavaLogFormatterClass**

An optional element to specify the class to use to format the log records. If no class is used, the logger uses the SimpleFormatter class. For more information about the SimpleFormatter class, see Chapter 15, “Custom data driver Java reference.”

LogDirectory

The absolute path of the directory to use for log files. If not specified, the default log directory is the log subdirectory under the ODA driver directory:

```
<Actuate_install_directory>/oda/<driver_dir>/log/.
```

LogFilenamePrefix

The prefix for log files. Log files are named in the following format:
 <prefix>-<date and time>.log.

LogLevel

The lowest level of log records to publish. For a list of log levels, see the fields defined for Class Level in Chapter 15, “Custom data driver Java reference.”

VendorInfo

Specifies information about the ODA driver and its vendor. e.Report Designer Professional does not use this information.

Schema

```

<xs:element name="VendorInfo">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="ProductName" type="xs:string"/>
      <xs:element name="Version" type="xs:string"/>
      <xs:element name="Vendor" type="xs:string"/>
      <xs:element name="VendorURL" type="xs:string"
        minOccurs="0"/>
      <xs:element name="VendorPhone" type="xs:string"
        minOccurs="0"/>
      <xs:element name="CopyrightNotice" type="xs:string"
        minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

```

Elements **ProductName**

Required element that provides the name of the custom ODA driver

Version

Required element that provides the version of the custom ODA driver

Vendor

Required element that provides the name of the company or other organization that created the custom ODA driver

VendorURL

Optional element to provide a URL for the vendor

VendorPhone

Optional element to provide a phone number for the vendor

CopyrightNotice

Optional element to provide a copyright notice for the custom ODA driver

14

Custom data driver XML reference

This chapter contains the following topics:

- Communicating the structure of a custom data source
- Data source builder reference

Communicating the structure of a custom data source

A report developer uses a custom driver in e.Report Designer Professional to access a data source. The custom driver must provide a data source builder application that provides information about the data source and supports the report developer in specifying the parameters and properties of a data stream component.

e.Report Designer Professional uses XML files and a request-and-response format to communicate with the data source builder. The data source builder parses XML request files and creates XML response files.

This chapter is a reference guide to the XML schema for this communication. These XML elements define the structure, content, and semantics of the XML request and response streams. Your data source builder must recognize and use these XML elements.

Data source builder reference

This section provides an alphabetical list of the XML elements that the data source builder uses.

ArrayOfInputFieldDefinition

List of input parameter field definitions

Schema

```
<xs:complexType name="ArrayOfInputFieldDefinition">
  <xs:sequence>
    <xs:element name="InputFieldDefinition"
      type="actu:InputFieldDefinition" maxOccurs="unbounded" />
  </xs:sequence>
</xs:complexType>
```

Elements **InputFieldDefinition**
The properties of an input parameter field

See also [InputFieldDefinition](#)

ArrayOfInputParameterDefinition

List of input parameter definitions

Schema	<pre><xs:complexType name="ArrayOfInputParameterDefinition"> <xs:sequence> <xs:element name="InputParameterDefinition" type="actu:InputParameterDefinition" maxOccurs="unbounded" /> </xs:sequence> </xs:complexType></pre>
Elements	InputParameterDefinition The properties of an input parameter
See also	InputParameterDefinition

ArrayOfNameValuePair

List of NameValuePair elements

Schema	<pre><xs:complexType name="ArrayOfNameValuePair"> <xs:sequence> <xs:element name="NameValuePair" type="actu:NameValuePair" maxOccurs="unbounded" /> </xs:sequence> </xs:complexType></pre>
Elements	NameValuePair A NameValuePair element
See also	NameValuePair

ArrayOfOutputFieldDefinition

List of output parameter field definitions

Schema	<pre><xs:complexType name="ArrayOfOutputFieldDefinition"> <xs:sequence maxOccurs="unbounded"> <xs:element name="OutputFieldDefinition" type="actu:OutputFieldDefinition" /> </xs:sequence> </xs:complexType></pre>
Elements	OutputFieldDefinition The properties of an output parameter field
See also	OutputFieldDefinition

ArrayOfOutputParameterDefinition

List of output parameter definitions

Schema `<xs:complexType name="ArrayOfOutputParameterDefinition">
 <xs:sequence maxOccurs="unbounded">
 <xs:element name="OutputParameterDefinition"
 type="actu:OutputParameterDefinition" />
 </xs:sequence>
</xs:complexType>`

Elements **OutputParameterDefinition**
The properties of an output parameter

See also [OutputParameterDefinition](#)

ArrayOfProperty

List of property name-value pairs

Schema `<xs:complexType name="ArrayOfProperty">
 <xs:sequence>
 <xs:element name="Property" type="actu:Property"
 maxOccurs="unbounded" />
 </xs:sequence>
</xs:complexType>`

Elements **Property**
A name-value pair that describes an object

See also [Property](#)

ArrayOfResultColumn

List of result columns in each row of the result set

Schema `<xs:complexType name="ArrayOfResultColumn">
 <xs:sequence>
 <xs:element name="ResultColumn" type="actu:ResultColumn"
 maxOccurs="unbounded" />
 </xs:sequence>
</xs:complexType>`

Elements **ResultColumn**
The properties of a result set column

See also [ResultColumn](#)

ArrayOfString

List of String elements

Schema

```
<xs:complexType name="ArrayOfString">
  <xs:sequence>
    <xs:element name="String" type="xs:string"
      maxOccurs="unbounded" />
  </xs:sequence>
</xs:complexType>
```

Elements **String**
A String value

AxisType

Axis type of a result set column. AxisType has one of the following values:

- Dimension Member
- Dimension Attribute
- Measure

Information objects use AxisType to provide information to report developers and applications about how to use the column.

Schema

```
<xs:simpleType name="AxisType" final="restriction">
  <xs:restriction base="xs:string">
    <xs:enumeration value="DimensionMember"/>
    <xs:enumeration value="DimensionAttribute"/>
    <xs:enumeration value="Measure"/>
  </xs:restriction>
</xs:simpleType>
```

Elements **DimensionAttribute**
Specifies that the column is a dimension attribute

DimensionMember
Specifies that the column is a dimension member

Measure
Specifies that the column is a measure

ColumnAxisInfo

Multidimensional attributes of a result set column

Schema

```
<xs:complexType name="ColumnAxisInfo">
  <xs:all>
    <xs:element name="AxisType" type="actu:AxisType">
    </xs:element>
    <xs:element name="OnColumnLayout" type="xs:boolean"
      minOccurs="0">
    </xs:element>
  </xs:all>
</xs:complexType>
```

AxisType

Axis type of a result set column

OnColumnLayout

The layout to use in a report. If true, the layout is as a column. This information is used by controls such as a crosstab to present a default layout. The default value is true.

See also AxisType

ConnectionProperties

Properties of the open data source connection

Schema

```
<xs:complexType name="ConnectionProperties">
  <xs:complexContent>
    <xs:extension base="actu:ArrayOfProperty" />
  </xs:complexContent>
</xs:complexType>
```

See also ArrayOfProperty
Property

DataStreamDefinition

The properties that define the data stream to use in a report design. DataStreamDefinition is an element of both DesignSessionRequest and DesignSessionResponse.

Schema

```
<xs:complexType name="DataStreamDefinition">
  <xs:all>
    <xs:element name="Name" type="xs:string"></xs:element>
    <xs:element name="Type" type="xs:string"></xs:element>
    <xs:element name="ShortDescription" type="xs:string"
      minOccurs="0">
    </xs:element>
  </xs:all>
</xs:complexType>
```

```

<xs:element name="Command" type="xs:string"></xs:element>
<xs:element name="PublicProperties"
  type="actu:ArrayOfProperty" minOccurs="0">
</xs:element>
<xs:element name="PrivateProperties"
  type="actu:ArrayOfProperty" minOccurs="0">
</xs:element>
<xs:element name="InputParameters"
  type="actu:ArrayOfInputParameterDefinition" minOccurs="0">
</xs:element>
<xs:element name="OutputParameters"
  type="actu:ArrayOfOutputParameterDefinition"
  minOccurs="0">
</xs:element>
<xs:element name="PrimaryResultSet"
  type="actu:ResultSetDefinition" minOccurs="0">
</xs:element>
</xs:all>
</xs:complexType>

```

Elements

Command

The command or query to execute to retrieve the result set. Command or query syntax is specific to the data source and must be recognized by the driver when the report runs. A command typically retrieves a result set but also can perform other tasks, such as updating a file.

InputParameters

Definitions of input parameters to map as report parameters. A report developer can edit the report parameters. For example, the report developer can change the definition of the parameter, its control type, and so on.

Name

The external name of the data stream

OutputParameters

Definitions of output parameters. Actuate Basic variables of the Data Source component store output parameter values.

PrimaryResultSet

The `ResultSetDefinition` for the result set that the data stream returns. The result set maps to a Data Row component.

PrivateProperties

Private properties of the custom data source. A report developer cannot edit these properties.

PublicProperties

Named properties of the open data source. A report developer can edit these properties. Each public property maps by name to a corresponding property in the AFC class for the data source. If the design session interface specifies a public

property that does not exist in the corresponding AFC class, e.Report Designer Professional ignores the property.

ShortDescription

A business description of the data stream

Type

The type of the data stream, which is specific to the data source provider.

See also [ArrayOfInputParameterDefinition](#)
 [ArrayOfOutputParameterDefinition](#)
 [ArrayOfProperty](#)
 [InputParameterDefinition](#)
 [OutputParameterDefinition](#)
 [Property](#)
 [ResultSetDefinition](#)

DesignSessionRequest

The request that defines the data that e.Report Designer Professional passes to a custom data source builder at the start of the design session

Schema

```
<xs:complexType name="DesignSessionRequest">
  <xs:all>
    <xs:element name="SessionLocale" type="actu:SessionLocale">
    </xs:element>
    <xs:element name="SessionEditMode"
      type="actu:SessionEditMode" minOccurs="0"/>
    <xs:element name="ExternalDesignState"
      type="actu:ExternalState" minOccurs="0">
    </xs:element>
    <xs:element name="PublicConnectionProperties"
      type="actu:ConnectionProperties" minOccurs="0" />
    <xs:element name="PrivateConnectionProperties"
      type="actu:ConnectionProperties" minOccurs="0" />
    <xs:element name="DataStreamDefinition"
      type="actu:DataStreamDefinition" minOccurs="0" />
  </xs:all>
</xs:complexType>
```

Elements **DataStreamDefinition**
 The properties of a data stream

ExternalDesignState

An optional element that retrieves the private state of the custom data source builder when the builder exits. The data source builder uses this information to return to the state of its last session.

PrivateConnectionProperties

Properties of the connection that a report developer cannot edit.

PublicConnectionProperties

Properties of the connection that a report developer can edit.

SessionEditMode

Specifies whether report developers can make changes using the custom data source builder.

SessionLocale

Specifies the application locale of e.Report Designer Professional. The application locale specifies the time and date format, currency format, and other characteristics of the data. The data source builder uses the same locale as e.Report Designer Professional, if possible. For example, if e.Report Designer Professional uses the Japanese locale, the data source builder uses the Japanese locale in its design session. If your data source builder code does not support the specified locale, it should display a message to the report developer that lists the supported locales. The report developer can then change the locale in e.Report Designer Professional to a supported locale.

See also ConnectionProperties
 DataStreamDefinition
 ExternalState
 SessionEditMode
 SessionLocale

DesignSessionResponse

Definition of the data stream and connection that a custom data source builder returns when the builder exits

Schema

```
<xs:complexType name="DesignSessionResponse">
  <xs:all>
    <xs:element name="ResponseState" type="actu:ResponseState">
    </xs:element>
    <xs:element name="ExternalDesignState"
      type="actu:ExternalState" minOccurs="0">
    </xs:element>
    <xs:element name="PublicConnectionProperties"
      type="actu:ConnectionProperties" minOccurs="0" />
    <xs:element name="PrivateConnectionProperties"
      type="actu:ConnectionProperties" minOccurs="0" />
    <xs:element name="DataStreamDefinition"
      type="actu:DataStreamDefinition" minOccurs="0" />
  </xs:all>
</xs:complexType>
```

Elements	DataStreamDefinition The properties of a data stream.
	ExternalDesignState An optional element that specifies the private state of the custom data source builder when it exits. The data source builder uses this information to return to the state of its last session.
	PrivateConnectionProperties The properties of the connection that a report developer cannot edit.
	PublicConnectionProperties The properties of the connection that a report developer can edit.
	ResponseState The state of the custom data source builder when it exits. Valid values are: ■ OK, indicating that the data source builder session succeeded. e.Report Designer Professional can consume and save the data. ■ UserCancelled, indicating that the report developer stopped the data source builder session. ■ LoginFailed, indicating that the data source builder is not able to establish a connection using ConnectionProperties in DesignSessionRequest.
See also	ConnectionProperties DataStreamDefinition ExternalState ResponseState

DynamicConditionReference

Description of the data source column that an ad hoc parameter filters

Schema `<xs:complexType name="DynamicConditionReference">
 <xs:all>
 <xs:element name="Identifier" type="xs:string"/>
 <xs:element name="IdentifierNativeTypeCode" type="xs:short"/>
 </xs:all>
</xs:complexType>`

Elements	Identifier The name of the referenced data source column.
	IdentifierNativeTypeCode The native data type of the referenced column in the data source.

DynamicQuery

Definition of the input parameter's dynamic query, if any. Specifying DynamicQuery enables e.Report Designer Professional to provide the report developers with a dynamic list of values from which to can select top-level scalar parameter values.

Schema

```
<xs:complexType name="DynamicQuery">
  <xs:all>
    <xs:element name="IsEnabled" type="xs:boolean" default="true"
      minOccurs="0"/>
    <xs:element name="QueryText" type="xs:string" minOccurs="0"/>
    <xs:element name="ValueColumn" type="xs:string"
      minOccurs="0"/>
    <xs:element name="DisplayNameColumn" type="xs:string"
      minOccurs="0"/>
  </xs:all>
</xs:complexType>
```

Elements **DisplayNameColumn**

The result set column name whose values provide localized descriptions of the values in valueColumn. If you provide a value for DisplayNameColumn, the user selects from the list of values in that column, and the query uses the ValueColumn value that appears in the same data row as the selected DisplayNameColumn value. For example, set valueColumn to the name of the column that contains company identifier codes for use in the query, and set DisplayNameColumn to the name of the column that provides the company names. By setting DisplayNameColumn, you enable users to pick from a list of company names instead of the corresponding codes used by the query.

IsEnabled

Specifies whether the dynamic query is enabled. The default value is true. If false, the QueryText remains but is not used.

QueryText

The Actuate SQL query.

ValueColumn

The result set column name whose selected value is used in the query. The ValueColumn must be one of the column names in the dynamic query's primary result set. If DisplayNameColumn is not set, then the user selects from the retrieved values of the ValueColumn column.

ExternalState

Private state of the previous data source builder session

Schema	<pre> <xs:complexType name="ExternalState"> <xs:all> <xs:element name="Version" type="xs:string"/> <xs:element name="StateInfo" type="actu:StateInfo"/> </xs:all> </xs:complexType> </pre>
Elements	StateInfo Private data formatted by the data source builder to describe the state of its last design session. Can be either String or binary format. Version The version of the external state's data format.
See also	StateInfo

HorizontalAlignment

HorizontalAlignment values are left, center, right, or automatic. The default value is Automatic. If the HorizontalAlignment value is automatic, e.Report Designer Professional determines the alignment based on its own default rules.

Schema	<pre> <xs:simpleType name="HorizontalAlignment" final="restriction"> <xs:restriction base="xs:string"> <xs:enumeration value="Automatic"/> <xs:enumeration value="Left"/> <xs:enumeration value="Center"/> <xs:enumeration value="Right"/> </xs:restriction> </xs:simpleType> </pre>
Elements	Automatic Specifies the default alignment of the e.Report Designer Professional control type. Center Specifies center alignment. Left Specifies left alignment. Right Specifies right alignment.

InputFieldDefinition

Definition of an input parameter field

Schema

```

<xs:complexType name="InputFieldDefinition">
  <xs:all>
    <xs:element name="Name" type="xs:string"/>
    <xs:element name="DisplayName" type="xs:string"
      minOccurs="0">
    <xs:element name="Description" type="xs:string"
      minOccurs="0">
    </xs:element>
    <xs:element name="DataType" type="actu:OdaScalarDataType">
    </xs:element>
    <xs:element name="IsHidden" type="xs:boolean" minOccurs="0">
    </xs:element>
    <xs:element name="IsRequired" type="xs:boolean"
      minOccurs="0">
    </xs:element>
    <xs:element name="DefaultValue" type="xs:string"
      minOccurs="0">
    </xs:element>
    <xs:element name="SelectValueList" type="actu:ArrayOfString"
      minOccurs="0">
    </xs:element>
    <xs:element name="SelectNameValueList"
      type="actu:ArrayOfNameValuePair" minOccurs="0">
    </xs:element>
    <xs:element name="FieldControlType" minOccurs="0">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:enumeration value="ControlList" />
          <xs:enumeration value="ControlListAllowNew" />
          <xs:enumeration value="ControlTextBox" />
        </xs:restriction>
      </xs:simpleType>
    </xs:element>
  </xs:all>
</xs:complexType>

```

Elements

DataType
The ODA scalar type of the input parameter field.

DefaultValue

The default value of the input parameter field. Use the XML data format that corresponds to the ODA data type of the input parameter field. For example, an input parameter with the timestamp data type should have the following format:

yyyy-mm-ddThh:mm:ss

Description

The description of the input parameter field.

DisplayName

The display name of the input parameter field.

FieldControlType

The type of control for the input parameter field. Valid values are:

- `ControlList`
- `ControlListAllowNew`
- `ControlTextBox`

IsHidden

Indicates whether the input parameter field is hidden or visible.

IsRequired

Indicates whether the input parameter field is required.

Name

A unique name for the input parameter field.

SelectValueList

Deprecated. Use `SelectNameValueList`.

SelectNameValueList

A list of selectable values and their corresponding display names for the input parameter field. For each value, use the XML data format that corresponds to the ODA data type of the input parameter field. For example, an input parameter of the timestamp data type should have the following format:

```
yyyy-mm-ddThh:mm:ss
```

See also `ArrayOfNameValuePair`
 `ArrayOfString`
 `NameValuePair`
 `OdaScalarDataType`

InputParameterDefinition

Definition of a scalar or complex input parameter. An input parameter consists of attributes such as the parameter name, data type, default value, and so on. A complex parameter also contains a collection of fields, which are defined in the `RecordDefinition` element.

Schema

```
<xs:complexType name="InputParameterDefinition">
  <xs:all>
    <xs:element name="Group" type="xs:string" minOccurs="0">
    </xs:element>
    <xs:element name="Name" type="xs:string"></xs:element>
```

```

<xs:element name="DataType" type="actu:OdaDataType">
</xs:element>
<xs:element name="DefaultValue" type="xs:string"
  minOccurs="0">
<xs:element name="Description" type="xs:string"
  minOccurs="0">
</xs:element>
<xs:element name="IsRequired" type="xs:boolean"
  minOccurs="0">
</xs:element>
<xs:element name="IsPassword" type="xs:boolean"
  minOccurs="0">
</xs:element>
<xs:element name="IsHidden" type="xs:boolean" minOccurs="0">
</xs:element>
<xs:element name="DisplayName" type="xs:string"
  minOccurs="0">
</xs:element>
<xs:element name="SelectValueList" type="actu:ArrayOfString"
  minOccurs="0">
</xs:element>
<xs:element name="SelectNameValueList"
  type="actu:ArrayOfNameValuePair" minOccurs="0">
</xs:element>
<xs:element name="DynamicValueList" type="actu:DynamicQuery"
  minOccurs="0">
</xs:element>
<xs:element name="DynamicCondition"
  type="actu:DynamicConditionReference" minOccurs="0">
</xs:element>
<xs:element name="ControlType" minOccurs="0">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="ControlRadioButton" />
      <xs:enumeration value="ControlList" />
      <xs:enumeration value="ControlListAllowNew" />
      <xs:enumeration value="ControlCheckBox" />
      <xs:enumeration value="ControlTextBox" />
    </xs:restriction>
  </xs:simpleType>
</xs:element>
<xs:element name="RecordDefinition"
  type="actu:ArrayOfInputFieldDefinition" minOccurs="0">
</xs:element>
</xs:all>
</xs:complexType>

```

Elements**ControlType**

The type of control. Valid values are:

- `ControlRadioButton`
- `ControlList`
- `ControlListAllowNew`
- `ControlCheckBox`
- `ControlTextBox`

DataType

The ODA data type of the input parameter.

DefaultValue

The default value of the input parameter. If a user can use the parameter to provide an expression that specifies additional conditions, provide the default value in QBE syntax. For information about the QBE syntax, see *Using Information Console*. If a user can provide only a specific value for the parameter, use the XML data format that corresponds to the ODA data type of the input parameter field. For example, an input parameter with the timestamp data type should have the following format:

`yyyy-mm-ddThh:mm:ss`

Description

The description of the input parameter field.

DisplayName

The display name of the input parameter.

DynamicCondition

An optional element that describes the data source column that is filtered by an ad hoc parameter. This element only applies to a scalar input parameter.

DynamicValueList

A dynamic list of values from which a user can select for a scalar input parameter.

Group

The parameter group to which the input parameter belongs.

IsHidden

Indicates whether the input parameter is hidden.

IsPassword

Indicates whether to mask the input parameter value.

IsRequired

Indicates whether the input parameter is required.

Name

The name of the input parameter.

RecordDefinition

The definition of complex input parameter fields.

SelectValueList

Deprecated. Use SelectNameValueList.

SelectNameValueList

A list of selectable values and their corresponding display names for the input parameter. If a user can use the parameter to provide an expression that specifies additional conditions, use QBE syntax to create each value. For information about QBE syntax, *Using Information Console*. If a user can provide only a specific value for the parameter, create each value using the XML data format that corresponds to the ODA data type of the input parameter. For example, an input parameter with the timestamp data type should have the following format:

```
yyyy-mm-ddThh:mm:ss
```

See also [ArrayOfInputFieldDefinition](#)
 [ArrayOfNameValuePair](#)
 [ArrayOfString](#)
 [DynamicConditionReference](#)
 [DynamicQuery](#)
 [InputFieldDefinition](#)
 [NameValuePair](#)
 [OdaDataType](#)

NameValuePair

The definition of a name and its corresponding value

Schema

```
<xs:complexType name="NameValuePair">
  <xs:all>
    <xs:element name="Name" type="xs:string" />
    <xs:element name="Value" type="xs:string" />
  </xs:all>
</xs:complexType>
```

Elements **Name**
 The name that corresponds to the value. For example, the name can be the display name for an input parameter value. The string can be empty.

Value

The value. The string can be empty.

OdaDataType

Open data access data types. The definition and range of values for each data type is specific to the programming language the ODA data source uses. For example, an ODA driver that is implemented in Java maps its native data types, which are expressed as `java.sql.Types`, to the corresponding ODA data type. e.Report Designer Professional maps and converts the ODA data types to one of its data types using the `DataTypeMapping` element of `odaconfig.xml`. For example, the native data type `DOUBLE` maps to the ODA data type `double`. You also can assign an alternate ODA data type to a native type, as shown in the following example:

```
<DataTypeMapping>
  <NativeDataType>DOUBLE</NativeDataType>
  <NativeDataTypeCode>8</NativeDataTypeCode>
  <OdaScalarDataType>Double</OdaScalarDataType>
  <AlternativeODADataTypes>
    <OdaScalarDataType>Integer</OdaScalarDataType>
    <OdaScalarDataType>String</OdaScalarDataType>
  </AlternativeODADataTypes>
</DataTypeMapping>
```

Schema

```
<xs:simpleType name="OdaDataType" final="restriction">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Date" />
    <xs:enumeration value="Double" />
    <xs:enumeration value="Integer" />
    <xs:enumeration value="String" />
    <xs:enumeration value="Structure" />
    <xs:enumeration value="Table" />
    <xs:enumeration value="Time" />
    <xs:enumeration value="Timestamp" />
    <xs:enumeration value="Decimal" />
    <xs:enumeration value="Blob" />
    <xs:enumeration value="Clob" />
  </xs:restriction>
</xs:simpleType>
```

- Elements**
- Blob**
Supports a binary large object
 - Clob**
Supports a character large object
 - Date**
Supports a date value

Decimal

Supports a decimal value

Double

Supports a double value

Integer

Supports an integer value

String

Supports a text string

Structure

Supports a structure value

Table

Supports a table value

Time

Supports a time value

Timestamp

Supports a timestamp value that contains both date and time information

OdaScalarDataType

Open data access scalar data types. The definition and range of values for each data type is specific to the programming language that the ODA data source uses.

Schema

```

<xs:simpleType name="OdaScalarDataType" final="restriction">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Date" />
    <xs:enumeration value="Double" />
    <xs:enumeration value="Integer" />
    <xs:enumeration value="String" />
    <xs:enumeration value="Time" />
    <xs:enumeration value="Timestamp" />
    <xs:enumeration value="Decimal"/>
    <xs:enumeration value="Blob"/>
    <xs:enumeration value="Clob"/>
  </xs:restriction>
</xs:simpleType>

```

Elements

Blob
Supports a binary large object

Clob
Supports a character large object

Date

Supports a date value

Decimal

Supports a decimal value

Double

Supports a double value

Integer

Supports an integer value

String

Supports a text string

Time

Supports a time value

Timestamp

Supports a timestamp value that contains both date and time information

OutputFieldDefinition

Definition of an output parameter field

Schema `<xs:complexType name="OutputFieldDefinition">`
 `<xs:all>`
 `<xs:element name="Name" type="xs:string">`
 `</xs:element>`
 `<xs:element name="Description" type="xs:string"`
 `minOccurs="0">`
 `</xs:element>`
 `<xs:element name="DisplayName" type="xs:string"`
 `minOccurs="0">`
 `</xs:element>`
 `<xs:element name="NativeTypeCode" type="xs:short">`
 `</xs:element>`
 `<xs:element name="NativeTypeName" type="xs:string">`
 `</xs:element>`
 `</xs:all>`
`</xs:complexType>`

Elements **Description**
 A description of the parameter

DisplayName

The display name of the output parameter or output parameter field

Name

The name of the output parameter or output parameter field

NativeTypeCode

The native type code of the output parameter or output parameter field. odaconfig.xml maps the native data type of the open source to an ODA data type. Then, e.Report Designer Professional uses the ODA data type to map to its own native data type. A value of 0 means that the open source native type is unknown.

NativeTypeName

The native type name of the output parameter or output parameter field

OutputParameterDefinition

Definition of a complex output parameter. A complex output parameter consists of attributes, such as the parameter name, data type, default value, and so on. It also contains a collection of fields, which are defined in the RecordDefinition element.

Schema

```
<xs:complexType name="OutputParameterDefinition">
  <xs:all>
    <xs:element name="Attributes"
      type="actu:OutputFieldDefinition">
    </xs:element>
    <xs:element name="RecordDefinition"
      type="actu:ArrayOfOutputFieldDefinition" minOccurs="0">
    </xs:element>
  </xs:all>
</xs:complexType>
```

Elements **RecordDefinition**
The definition of a complex output parameter field

See also [ArrayOfOutputFieldDefinition](#)
[OutputFieldDefinition](#)

Property

A property name-value pair

Schema

```
<xs:complexType name="Property">
  <xs:all>
    <xs:element name="Name" type="xs:string"></xs:element>
    <xs:element name="Value" type="xs:string"></xs:element>
  </xs:all>
</xs:complexType>
```

Elements	Name
	The property name
	Value
	The property value

ResponseState

Indicates to e.Report Designer Professional how to proceed after data source builder exits

Schema `<xs:simpleType name="ResponseState" final="restriction">
 <xs:restriction base="xs:string">
 <xs:enumeration value="Ok" />
 <xs:enumeration value="UserCancelled" />
 <xs:enumeration value="LoginFailed" />
 <xs:enumeration value="ProviderError" />
 </xs:restriction>
</xs:simpleType>`

Elements **LoginFailed**
Indicates that the data source builder is not able to establish a connection using the connection properties in DesignSessionRequest

Ok
Indicates that the data source builder session succeeded. e.Report Designer Professional can consume and save the data.

UserCancelled
Indicates that the report developer stopped the data source builder session

ProviderError
Indicates that a failure occurred that is unrelated to a login failure or a report developer cancelling the data source builder session

ResultColumn

Definition of a result set column

Schema `<xs:complexType name="ResultColumn">
 <xs:all>
 <xs:element name="Name" type="xs:string"/></xs:element>
 <xs:element name="Position" type="xs:unsignedShort"/>
 </xs:element>`

```

<xs:element name="DisplayName" type="xs:string"
  minOccurs="0">
</xs:element>
<xs:element name="NativeTypeCode" type="xs:short">
</xs:element>
<xs:element name="NativeTypeName" type="xs:string">
</xs:element>
<xs:element name="DisplayLength" type="xs:short"
  minOccurs="0">
</xs:element>
<xs:element name="DisplayFormat" type="xs:string"
  minOccurs="0">
</xs:element>
<xs:element name="TextFormat" default="Plain" minOccurs="0">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="Plain"/>
      <xs:enumeration value="HTML"/>
      <xs:enumeration value="RTF"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
<xs:element name="HorizontalAlignment"
  type="actu:HorizontalAlignment" minOccurs="0">
</xs:element>
<xs:element name="Wrap" default="None" minOccurs="0">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="None"/>
      <xs:enumeration value="Word"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
<xs:element name="Heading" type="xs:string" minOccurs="0">
</xs:element>
<xs:element name="Description" type="xs:string"
  minOccurs="0">
</xs:element>
<xs:element name="HelpText" type="xs:string" minOccurs="0">
</xs:element>
<xs:element name="Lineage" type="xs:string" minOccurs="0">
</xs:element>
<xs:element name="Axis" type="actu:ColumnAxisInfo"
  minOccurs="0">
</xs:element>
</xs:all>
</xs:complexType>

```

Elements**Axis**

The axis metadata information for a column retrieved from a multidimensional data source

Description

A description of the column

DisplayFormat

The preferred format in which to display this data in the report output

DisplayLength

The default display length of the data column. e.Report Designer Professional uses this value to set the default length of the corresponding data control. A value of -1 means that the length is unknown.

DisplayName

The business name of the column. e.Report Designer Professional uses this name as a default column label and as a reference name in the value expression of a data control.

Heading

The name of the column in the column header in the data source.

HelpText

The balloon help text for this column. A user can read this text for additional information about the column, for example when the user is selecting a value for a filter condition.

HorizontalAlignment

Horizontal alignment of the column value within the available display space.

Lineage

The original source of the column value. This value provides context information to help a report developer diagnose problems with the value. This element does not apply to derived values.

Name

The name of data row column. This is the name that the run-time driver recognizes. e.Report Designer Professional uses this name as an identifier in dialog boxes for sorting, filtering, and other tasks.

NativeTypeCode

The native data type code of the column. odaconfig.xml maps the native data type of the open source to an ODA data type. Then, e.Report Designer Professional uses the ODA data type to map to one of its data types. A value of 0 means that the native type is unknown.

NativeTypeName

The native data type name of the column. This name appears in such tools as the data row editor.

Position

The 1-based position of a column within the result set when the report or Actuate query runs.

TextFormat

Indicates whether the data column's string values are in plain text, HTML, or RTF format. The default value is Plain.

Wrap

Indicates whether to apply word wrapping. The default value is None. If None, there is no word wrapping. A value of Word enables word wrapping.

See also ColumnAxisInfo
HorizontalAlignment

ResultSetDefinition

The definition of a single result set that the data stream returns. Each **ResultSetDefinition** is a named collection of data columns. A result set maps to a Data Row component in e.Report Designer Professional. A data stream must have at least one result set. You define the result set in terms of data columns.

Schema

```
<xs:complexType name="ResultSetDefinition">
  <xs:all>
    <xs:element name="Name" type="xs:string"/></xs:element>
    <xs:element name="ResultSetColumns"
      type="actu:ArrayOfResultColumn">
      </xs:element>
    </xs:all>
  </xs:complexType>
```

Elements **Name**
The name of the result set

ResultSetColumns

A collection of data columns for this result set

See also ArrayOfResultColumn
ResultColumn

SessionEditMode

Specifies whether report developers can make changes using the custom data source builder. SessionEditMode can have one of the following values:

- Editable
The default value. This value indicates that e.Report Designer Professional will accept and use changes that a report developer makes using the custom data source builder.
- ReadOnly
This value indicates that e.Report Designer Professional will not use changes that a report developer makes using the custom data source builder. To avoid frustrating report developers, the custom data source builder should not permit a report developer to make changes when SessionEditMode value is ReadOnly.

Schema

```
<xs:simpleType name="SessionEditMode" final="restriction">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Editable"/>
    <xs:enumeration value="ReadOnly"/>
  </xs:restriction>
</xs:simpleType>
```

SessionLocale

The locale on the client machine that hosts e.Report Designer Professional. For example, if e.Report Designer Professional runs in Japanese, the custom data source builder uses the Japanese locale in its design session. SessionLocale can differ from the default locale set in e.Report Designer Professional. Not all data source builders honor all locales. For definitions of the values for SessionLocale, see *Working in Multiple Locales using Actuate Basic Technology*.

Schema

```
<xs:simpleType name="SessionLocale" final="restriction">
  <xs:restriction base="xs:string">
    <xs:enumeration value="ar_AE"/>
    <xs:enumeration value="ar_BH"/>
    <xs:enumeration value="ar_DZ"/>
    <xs:enumeration value="ar_EG"/>
    <xs:enumeration value="ar_IQ"/>
    <xs:enumeration value="ar_JO"/>
    <xs:enumeration value="ar_KW"/>
    <xs:enumeration value="ar_LB"/>
    <xs:enumeration value="ar_LY"/>
    <xs:enumeration value="ar_MA"/>
  </xs:restriction>
</xs:simpleType>
```



```

<xs:enumeration value="ar_OM"/>
<xs:enumeration value="ar_QA"/>
<xs:enumeration value="ar_SA"/>
<xs:enumeration value="ar_SY"/>
<xs:enumeration value="ar_TN"/>
<xs:enumeration value="ar_YE"/>
<xs:enumeration value="bg_BG"/>
<xs:enumeration value="cs_CZ"/>
<xs:enumeration value="da_DK"/>
<xs:enumeration value="de_AT"/>
<xs:enumeration value="de_CH"/>
<xs:enumeration value="de_DE"/>
<xs:enumeration value="de_LI"/>
<xs:enumeration value="el_GR"/>
<xs:enumeration value="en_AU"/>
<xs:enumeration value="en_BZ"/>
<xs:enumeration value="en_CA"/>
<xs:enumeration value="en_GB"/>
<xs:enumeration value="en_IE"/>
<xs:enumeration value="en_NZ"/>
<xs:enumeration value="en_US"/>
<xs:enumeration value="en_ZA"/>
<xs:enumeration value="es_ES"/>
<xs:enumeration value="es_MX"/>
<xs:enumeration value="et_EE"/>
<xs:enumeration value="fa_IR"/>
<xs:enumeration value="fi_FI"/>
<xs:enumeration value="fr_CA"/>
<xs:enumeration value="fr_CH"/>
<xs:enumeration value="fr_FR"/>
<xs:enumeration value="he_IL"/>
<xs:enumeration value="hr_HR"/>
<xs:enumeration value="hu_HU"/>
<xs:enumeration value="id_ID"/>
<xs:enumeration value="in_ID"/>
<xs:enumeration value="it_CH"/>
<xs:enumeration value="it_IT"/>
<xs:enumeration value="iw_IL"/>
<xs:enumeration value="ja_JP"/>
<xs:enumeration value="ko_KR"/>
<xs:enumeration value="lv_LV"/>
<xs:enumeration value="nl_BE"/>
<xs:enumeration value="nl_NL"/>
<xs:enumeration value="no_NO"/>
<xs:enumeration value="no_NY"/>
<xs:enumeration value="pl_PL"/>
<xs:enumeration value="pt_BR"/>

```

```

<xs:enumeration value="pt_PT"/>
<xs:enumeration value="ro_RO"/>
<xs:enumeration value="ru_RU"/>
<xs:enumeration value="sk_SK"/>
<xs:enumeration value="sl_SI"/>
<xs:enumeration value="sq_AL"/>
<xs:enumeration value="sr_YU"/>
<xs:enumeration value="sv_FI"/>
<xs:enumeration value="sv_SE"/>
<xs:enumeration value="th_TH"/>
<xs:enumeration value="tr_TR"/>
<xs:enumeration value="uk_UA"/>
<xs:enumeration value="zh_CN"/>
<xs:enumeration value="zh_HK"/>
<xs:enumeration value="zh_SG"/>
<xs:enumeration value="zh_TW"/>
</xs:restriction>
</xs:simpleType>

```

StateInfo

The state of the previous data source builder session

Schema `<xs:complexType name="StateInfo">`
 `<xs:choice>`
 `<xs:element name="StateInfoString" type="xs:string">`
 `</xs:element>`
 `<xs:element name="StateInfoBlob" type="xs:base64Binary">`
 `</xs:element>`
 `</xs:choice>`
`</xs:complexType>`

Elements **StateInfoString**
 State information in String format

StateInfoBlob

State information in binary format. Encode the string in base64 format. For more information about the base64 encoding scheme, see RFC 2045 at <http://www.ietf.org/rfc/rfc2045.txt>.

15

Custom data driver Java reference

This chapter contains the following topics:

- About Java interfaces and Java classes for creating a custom data driver
- Open data access Java reference

About Java interfaces and Java classes for creating a custom data driver

This chapter lists the Java interfaces and classes that implement run-time open data access (ODA) functionality. Use these interfaces and classes to create a custom driver.

The ODA run-time interface is modeled after JDBC so that its usage model is intuitive to a Java driver developer who is familiar with JDBC. The interface covers a subset of JDBC capabilities that suit reporting applications. The interface excludes JDBC features related to write and update operations, java.io stream processing, complex result set types and concurrency modes, and so on.

The ODA run-time interfaces support capabilities that are not defined in the JDBC model. The primary extended capabilities include:

- Access to result sets by name.
- Use of complex, non-scalar parameters, including the ability to use structure and table types as input and output parameters.
- Support for any type of data source.
- Support for using a statement with any query command text. This functionality is not limited to SQL statement syntax.
- Using the same connection to create different types of data source-specific statements.
- Assigning statement-specific properties before execution.

All ODA interface indexes are 1-based. Because Java data structures are typically 0-based, the ODA driver must be able to translate between 0-based and 1-based indexes.

The class that is the entry point to the ODA driver must implement the `IDriver` interface. Methods defined by this interface provide access to parameters, statements, and result sets. Actuate products use calls to `IConnectionMetaData`, `IDataSourceMetaData`, and `IParameterMetaData` methods to determine the functionality of the driver.

Each interface or class entry in this chapter includes a general description of the interface or class and a summary of its fields and methods. A detailed listing of methods for each interface or class includes the syntax, parameters, any return value, and whether the method throws an exception. Some methods are optional. If you do not want to implement an optional method, just declare the method, and throw the `UnsupportedOperationException`. The message of the `UnsupportedOperationException` should contain at least the name of the class and method to provide the context of the exception, as shown in the following example:

```

public class MyStatementClass extends IStatement
{
    public void setInt( int parameterId, int value )
    {
        // is not supported by this Statement class
        throw new UnsupportedOperationException
            "MyStatementClass.setInt( int parameterId, int value )" );
    }
}

```

For an example of an ODA driver that includes the configuration file, data source builder, and run-time classes, see the open data access flat file example that is installed in \Actuate11\oda\Examples\Flat File Example.

Open data access Java reference

This section provides an alphabetical list of the open data access Java interfaces and classes.

Class FileHandler

Syntax public class FileHandler extends StreamHandler

Description FileHandler is a class that publishes LogRecords to a specified file. LogManager.getLogger() sets a new FileHandler if the log directory has changed or the log file name has changed.

Constructors

Syntax public FileHandler(java.lang.String filename)

Constructs a FileHandler object to publish LogRecords to the specified file. The LogRecords are formatted using the default SimpleFormatter class. Calling publish() creates the physical file and any applicable parent directories.

Parameter **filename**
The file in which the LogRecords are published

Syntax public FileHandler(java.lang.String filename, LogFormatter formatter)

Constructs a FileHandler object to publish LogRecords to the specified, pre-existing file. The LogRecords are formatted using the specified LogFormatter. If the FileHandler instance exists, calling publish() creates the physical file and any applicable parent directories.

Parameters **filename**
The file in which the LogRecords are published

formatter
The LogFormatter to use to format the LogRecords

Method summary

Table 15-1 lists the methods for class FileHandler.

Table 15-1 Methods for Class FileHandler

Method	Description
close()	Closes the current FileHandler
publish()	Publishes a record

Methods inherited from class com.actuate.oda.util.logging.StreamHandler

finalize, flush, isLoggable, setFormatter, setOutputStream

Methods inherited from class `com.actuate.oda.util.logging.Handler`

`getFilter`, `getFormatter`, `getLevel`, `getLoggingErrorHandler`, `reportError`, `setFilter`, `setLevel`, `setLogginErrorHandler`

Methods inherited from class `java.lang.Object`

`clone`, `equals`, `getClass`, `hashCode`, `notify`, `notifyAll`, `toString`, `wait`, `wait`, `wait`

FileHandler.close

Syntax `public void close()`

Closes the current FileHandler

FileHandler.publish

Syntax `public void publish(LogRecord record)`

Publishes a record to the log file. If necessary, it creates the log file and parent directories before it publishes a record to the log file.

Parameter **record**

The record to publish to the log file

Interface Filter

Syntax public interface Filter

Description Use a Filter to provide more control over what is logged. Each Handler object can have an associated Filter. The Handler calls `isLoggable()` to check whether to publish a record.

Method summary

Table 15-2 describes the method for interface Filter.

Table 15-2 Method for interface Filter

Method	Description
<code>isLoggable()</code>	Checks whether to publish the record

Filter.isLoggable

Syntax public boolean isLoggable(LogRecord record)

Description Checks whether to publish the record

Parameter **record**
The log record to check

Returns True if the log record should be published

Class Handler

- Syntax** `public abstract class Handler extends Object`
- Description** Handler is an abstract class that obtains records from a Logger and outputs them to a specified destination. All log handlers that publish records to a console, file, or other destinations extend this class. For example, StreamHandler extends this class to publish records to a stream.

Constructor

- Syntax** `public Handler( )`
Constructs a Handler.

Method summary

Table 15-3 lists the methods for class Handler.

Table 15-3 Methods for class Handler

Method	Description
<code>close()</code>	Closes the current Handler and frees resources
<code>flush()</code>	Flushes the buffered output
<code>getFilter()</code>	Gets the Filter associated with this Handler
<code>getFormatter()</code>	Gets the LogFormatter associated with this Handler
<code>getLevel()</code>	Gets the Level associated with this Handler
<code>getLoggingErrorHandler()</code>	Gets the LoggingErrorHandler associated with this Handler
<code>isLoggable()</code>	Checks whether to log the specified record
<code>publish()</code>	Formats and publishes a record
<code>reportError()</code>	Reports an error to the associated LoggingErrorHandler
<code>setFilter()</code>	Sets the Filter to use for this Handler
<code>setFormatter()</code>	Sets the LogFormatter to use for this Handler
<code>setLevel()</code>	Sets the Level to use for this Handler
<code>setLoggingErrorHandler()</code>	Sets the LoggingErrorHandler to use for this Handler

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Handler.close

Syntax public void close()
 Closes the current Handler

Handler.flush

Syntax public abstract void flush()
 Flushes the buffered output

Handler.getFilter

Syntax public Filter getFilter()
 Gets the Filter associated with this Handler
Returns The associated Filter instance

Handler.getFormatter

Syntax public LogFormatter getFormatter()
 Gets the LogFormatter associated with this Handler
Returns The associated LogFormatter instance

Handler.getLevel

Syntax public Level getLevel()
 Gets the log level set for the Handler. The log level limits the records that are published.
Returns The level instance for this Handler

Handler.getLoggingErrorHandler

Syntax `public LoggingErrorHandler getLoggingErrorHandler( )`

Gets the error handler associated with this Handler

Returns A LoggingErrorHandler instance

Handler.isLoggable

Syntax `public boolean isLoggable(LogRecord record)`

Checks whether to log the specified record by determining that the record has a high enough log level, passes the associated Filter, and passes any other checks that the Handler specifies.

Parameter **record**
The log record to check

Returns True to log the log record

Handler.publish

Syntax `public void publish(LogRecord record)`

Publishes a record to the appropriate destination if it has a sufficiently high log level, and passes any associated Filter. Publish() also formats the record, if necessary.

Parameter **record**
The log record to format and publish

Handler.reportError

Syntax `public void reportError(java.lang.String message, java.lang.Exception exception, int errorCode)`

Reports an error to the associated LoggingErrorHandler

Parameters **errorCode**
The error code of the error

exception
The exception that caused the error

message

The error message

Handler.setFilter

Syntax public void setFilter(Filter filter)

Sets the Filter for this Handler

Parameters **filter**
The filter to set

Handler.setFormatter

Syntax public void setFormatter(LogFormatter formatter)

Sets the formatter that will format the log records

Parameter **formatter**
The formatter to set

Handler.setLevel

Syntax public void setLevel(Level level)

Sets the log level to limit the records that this Handler publishes

Parameter **level**
The log level to set

Handler.setLoggingErrorHandler

Syntax public void setLoggingErrorHandler(LoggingErrorHandler errorHandler)

Sets the LoggingErrorHandler for this Handler

Parameter **errorHandler**
The error handler to use for this Handler

Interface IBlob

Syntax public interface IBlob

Description IBlob is an interface for processing a binary large object (BLOB) data value. You need to implement this interface only if your ODA driver supports the BLOB data type. The methods in this interface require getting a handle for the IBLOB object. If an ODA result set column is a BLOB, use `IRResultSet.getBlob()` to obtain a handle for the IBLOB object. If an ODA output parameter is a BLOB, use `ICallStatement.getBlob()` to obtain a handle for the IBLOB object. An ODA input parameter does not support using a BLOB.

Method summary

Table 15-4 lists the methods for interface IBlob.

Table 15-4 Methods for interface IBlob

Method	Description
<code>getBinaryStream()</code>	Returns the entire BLOB value as a Java input stream
<code>getBytes()</code>	Returns a Java input stream that contains a part of the BLOB value, starting from the specified position in the BLOB and continuing for the specified number of bytes
<code>length()</code>	Returns the number of bytes in the BLOB value

IBlob.getBinaryStream

Syntax public `InputStream` `getBinaryStream()` throws `OdaException`

Returns a handle to the entire BLOB value

Returns The contents of the BLOB as a Java input stream of uninterpreted bytes

Throws `OdaException` if a data source error occurs

IBlob.getBytes

Syntax public `byte[]` `getBytes( long position, int length )` throws `OdaException`

An optional method that returns part of the BLOB value, starting from the position indicated by the position parameter, and continuing for the number of bytes that the length parameter specifies. If the BLOB contains fewer bytes after the specified position than the number that the length parameter requires, `getBytes()` returns the BLOB value up to the end of the BLOB.

IBlob.length

This method has a default implementation to provide this functionality when you do not implement this method explicitly. Consider implementing this method if you can optimize processing the BLOB based on your knowledge of the type of data that it contains.

Parameters	length
	The number of bytes to return. Using a negative value for length returns all bytes through the end of the BLOB.
	position
	The position of the starting byte. The first byte in the BLOB is at position 1.
Returns	A Java byte array that contains a specific part of the contents of the BLOB that is identified by its position
Throws	OdaException if a data source error occurs.

IBlob.length

Syntax	public long length() throws OdaException
	An optional method that returns the number of bytes in a BLOB
Returns	The number of bytes in a BLOB
Throws	OdaException if a data source error occurs

Interface ICallStatement

Syntax public interface ICallStatement extends IStatement

Description ICallStatement is an interface for callable statements. ICallStatement implements the mandatory methods that it inherits from IStatement, except executeQuery(). ICallStatement is required only if the ODA driver supports:

- Accessing result sets by name
- Using complex input or output parameters

All callable statement implementations, such as stored procedures and R/3 BAPIs, implement this interface.

Call ICallStatement interface methods only after calling IStatement.prepare().

Method summary

Table 15-5 lists the methods for interface ICallStatement.

Table 15-5 Methods for interface ICallStatement

Method	Description
findOutParameter()	Returns the 1-based index of the specified output scalar or structure parameter
getBigDecimal()	Returns the decimal value from the specified output parameter
getBlob()	Returns the BLOB value from the specified output parameter
getClob()	Returns the CLOB value from the specified output parameter
getDate()	Returns the java.sql.Date value from the specified output parameter
getDouble()	Returns the double value from the specified output parameter
getInt()	Returns the integer value from the specified output parameter
getMetaDataOf()	Returns the metadata of the specified result
getResultSet()	Returns the named result as an IResultSet object instance or returns null
getResultSetNames()	An optional method that returns the names of result sets that this ICallStatement can return

(continues)

Table 15-5 Methods for interface ICallStatement (continued)

Method	Description
getRow()	An optional method that returns the structure value from the specified output parameter
getSortSpec()	Returns the associated SortSpec instance specifying how to sort the result set columns
getString()	Returns the String value from the specified output parameter
getTime()	Returns the java.sql.Time value from the specified output parameter
getTimestamp()	Returns the java.sql.Timestamp value from the specified output parameter
setNewRow()	An optional method that returns an instance of IRowSet that represents the specified single row input parameter
setNewRowSet()	An optional method that returns an instance of IRowSet that represents the specified table input parameter
setSortSpec()	Specifies the SortSpec instance to determine sorting of the result set columns
wasNull()	Specifies whether the last output parameter a get method reads is null

Methods inherited from interface com.actuate.oda.IStatement

close, execute, execute, executeQuery, executeQuery, findInParameter, getMaxRows, getMetaData, getMetaData, getMoreResults, getParameterMetaData, getParameterType, getParameterType, getResultSet, getSortSpec, prepare, setBigDecimal, setBigDecimal, setDate, setDate, setDouble, setDouble, setInt, setInt, setMaxRows, setProperty, setPropertyInfo, setSortSpec, setString, setString, setTime, setTime, setTimestamp

ICallStatement.findOutParameter

Syntax public int findOutParameter(java.lang.String parameterName)
 throws OdaException

Returns the 1-based index of the specified output scalar or structure parameter. Implement this method when the driver supports output parameters and named parameters.

Parameters	parameterName The name of the output parameter
Returns	The index of the output parameter
Throws	OdaException if a data source error occurs

ICallStatement.getBigDecimal

Syntax	public BigDecimal getBigDecimal(int parameterId) throws OdaException Returns the decimal value from the output parameter at the specified position
Parameter	parameterId The 1-based index position of the parameter
Returns	The decimal value
Throws	OdaException if a data source error occurs
Syntax	public BigDecimal getBigDecimal(java.lang.String parameterName) throws OdaException Returns the decimal value from the output parameter with the specified name
Parameter	parameterName Name of the parameter
Returns	The decimal value
Throws	OdaException if a data source error occurs

ICallStatement.getBlob

Syntax	public IBlob getIBlob(int index) throws OdaException Returns BLOB value in an output parameter, using the position to identify the parameter. After using ICallStatement.getBlob(), an ODA driver must keep the IBLOB object and the BLOB's data accessible until the driver closes the result set.
Parameters	parameterId The 1-based index of the parameter
Returns	The BLOB value. Returns null if the parameter has a null value.
Throws	OdaException if a data source error occurs
Syntax	public IBlob getIBlob(java.lang.String columnName) throws OdaException Returns the BLOB value in an output parameter, using a name to identify the parameter. After using ICallStatement.getBlob(), an ODA driver must keep the IBLOB object and the BLOB's data accessible until the driver closes the result set.

Parameter	parameterName Name of the parameter
Returns	The BLOB value. Returns null if the parameter has a null value.
Throws	OdaException if a data source error occurs

ICallStatement.getClob

Syntax `public IClob getIClob(int index) throws OdaException`

Returns CLOB value in an output parameter, using the parameter position to identify the desired parameter. After using `ICallStatement.getClob( )`, an ODA driver must keep the ICLOB object and the CLOB's data accessible until the driver closes the result set.

Parameter **parameterId**
The 1-based index of the parameter

Returns The CLOB that represents the value. Returns null if the parameter has a null value.

Throws OdaException if a data source error occurs

Syntax `public IClob getIClob(java.lang.String columnName) throws OdaException`

Returns CLOB value in an output parameter, using the parameter name to identify the desired parameter. After using `ICallStatement.getClob( )`, an ODA driver must keep the ICLOB object and the CLOB's data accessible until the driver closes the result set.

Parameter **parameterName**
Name of the parameter

Returns The CLOB value. Returns null if the parameter has a null value.

Throws OdaException if a data source error occurs

ICallStatement.getDate

Syntax `public java.sql.Date getDate(java.lang.String parameterName)
 throws OdaException`

Returns the `java.sql.Date` value for a named output parameter. Implement this method when the driver supports output parameters, named parameters, and the Date data type.

Parameter **parameterName**
The name of the parameter

Returns The `java.sql.Date` value of the named parameter

Throws OdaException if a data source error occurs

Syntax public Date getDate(int parameterId)
throws OdaException

Returns the java.sql.Date value of a specified output parameter. Implement this method when the driver supports output parameters and the Date data type.

Parameters **parameterId**
The 1-based index of the parameter

Returns The java.sql.Date value of the parameter

Throws OdaException if a data source error occurs

ICallStatement.getDouble

Syntax public double getDouble(java.lang.String parameterName) throws OdaException

Returns the double value of a named output parameter. Implement this method when the driver supports output parameters, named parameters, and the Double data type.

Parameter **parameterName**
The name of the output parameter

Returns The double value

Throws OdaException if a data source error occurs

Syntax public double getDouble(int parameterId)
throws OdaException

Returns the double value of a specified output parameter. Implement this method when the driver supports output parameters and the double data type.

Parameter **parameterId**
The 1-based index of the parameter

Returns The double value

Throws OdaException if a data source error occurs

ICallStatement.getInt

Syntax public int getInt(java.lang.String parameterName) throws OdaException

Returns the integer value of a named output parameter. Implement this method when the driver supports output parameters, named parameters, and the integer data type.

Parameter	parameterName The name of the parameter
Returns	The integer value
Throws	OdaException if a data source error occurs
Syntax	<pre>public int getInt(int parameterId) throws OdaException</pre> Returns the integer value of a specified output parameter. Implement this method when the driver supports output parameters and the integer data type.
Parameter	parameterId The 1-based index of the parameter
Returns	The integer value
Throws	OdaException if a data source error occurs

ICallStatement.getMetaDataOf

Syntax	<pre>public IResultSetMetaData getMetaDataOf(String resultSetName) throws OdaException</pre> Returns an instance of IResultSetMetaData. Implement this method when the driver supports named result sets.
Parameter	resultSetNames The name of the result set
Returns	The metadata of the specified result
Throws	OdaException if a data source error occurs

ICallStatement.getResultSet

Syntax	<pre>public com.actuate.oda.IResultSet getResultSet(java.lang.String resultSetName) throws OdaException</pre> Returns the specified result as an IResultSet object instance, or returns null. Call this method once for each result set.
Parameter	resultSetNames The name of the target result set
Returns	An IResultSet object instance or null
Throws	OdaException if a data source error occurs

ICallStatement.getResultSetNames

- Syntax** `public java.lang.String[] getResultSetNames( ) throws OdaException`
 An optional method that returns the names of result sets that the current ICallStatement can return.
- Returns** An array of result set names
- Throws** OdaException if a data source error occurs

ICallStatement.getRow

- Syntax** `public com.actuate.oda.IRowSet getRow(java.lang.String parameterName) throws OdaException`
 An optional method that returns the structure value of a named output parameter. Does not return table structures. Implement this method when the driver supports named complex output parameters.
- Parameter** **parameterName**
 The name of the output parameter
- Returns** The structure value
- Throws** OdaException if a data source error occurs
- Syntax** `public com.actuate.oda.IRowSet getRow(int parameterId) throws OdaException`
 An optional method that returns the structure value of a specified output parameter. Does not return table structures. Implement this method when the driver supports complex output parameters.
- Parameter** **parameterId**
 The 1-based index of the parameter
- Returns** The structure value
- Throws** OdaException if data source error occurs

ICallStatement.getSortSpec

- Syntax** `public SortSpec getSortSpec(java.lang.String resultSetName) throws OdaException`
 Returns the associated SortSpec instance that specifies how to sort the result set columns

Parameter	resultSetNames The name of the result set
Returns	The SortSpec associated with the specified result set or null if no SortSpec is associated
Throws	OdaException if a data source error occurs

ICallStatement.getString

Syntax `public java.lang.String getString(java.lang.String parameterName)`
 throws OdaException

Returns the String value of a named output parameter. You can use this method to return a String representation of a non-String type parameter. The format of the returned string is implementation-dependent. Implement this method when the driver supports output parameters, named parameters, and the String data type.

Parameter **parameterName**
 The name of the parameter

Returns The String value

Throws OdaException if data source error occurs

Syntax `public java.lang.String getString(int parameterId) throws OdaException`

Returns the String value of a specified output parameter. You can use this method to return the String representation of a non-String type parameter. The format of the returned string is implementation-dependent. Implement this method when the driver supports output parameters and the String data type.

Parameter **parameterId**
 The 1-based index of the parameter

Returns The String value of the specified output parameter

Throws OdaException if a data source error occurs

ICallStatement.getTime

Syntax `public java.sql.Time getTime(java.lang.String parameterName)`
 throws OdaException

Returns the java.sql.Time value of a named output parameter. Implement this method when the driver supports output parameters, named parameters, and the Time data type.

Parameter **parameterName**
 The name of the parameter

Returns The Time value of the parameter

Throws OdaException if a data source error occurs

Syntax `public java.sql.Time getTime(int parameterId)`
throws OdaException

Returns the java.sql.Time value of a specified output parameter. Implement this method when the driver supports output parameters and the Time data type.

Parameter **parameterId**
The 1-based index of the parameter

Returns The Time value of the parameter

Throws OdaException if a data source error occurs

ICallStatement.getTimestamp

Syntax `public java.sql.Timestamp getTimestamp(java.lang.String parameterName)`
throws OdaException

Returns the java.sql.Timestamp value of a named output parameter. Implement this method when the driver supports output parameters, named parameters, and the Timestamp data type.

Parameter **parameterName**
The name of the parameter

Returns The Timestamp value

Throws OdaException if data source error occurs

Syntax `public java.sql.Timestamp getTimestamp(int parameterId)`
throws OdaException

Returns the java.sql.Timestamp value of a specified output parameter. Implement this method when the driver supports output parameters and the Timestamp data type.

Parameter **parameterId**
The 1-based index of the parameter

Returns The Timestamp value

Throws OdaException if a data source error occurs

ICallStatement.setNewRow

Syntax `public com.actuate.oda.IRowSet setNewRow(java.lang.String parameterName)`
throws OdaException

An optional method that returns an instance of IRowSet that represents the specified single row input parameter. Implement this method when the driver supports named complex input parameters. The client calls the IRowSet setter methods to populate the input parameter. For example:

```
IRowSet myStruct = myStatement.setNewRow( "MyStructureName" );
myStruct.next();
myStruct.setString( 1, "myValue" );
```

- Parameter** **parameterName**
The name of the input parameter
- Returns** An IRowSet instance
- Throws** OdaException if data source error occurs
- Syntax** `public com.actuate.oda.IRowSet setNewRow(int parameterId)
throws OdaException`

An optional method that returns an instance of IRowSet that represents the specified single row input parameter. Implement this method when the driver supports complex input parameters. The client calls IRowSet setter methods to populate the input parameter. For example:

```
IRowSet myStruct = myStatement.setNewRow( 1 );
myStruct.next();
myStruct.setString( 1, "myValue" );
```

- Parameter** **parameterId**
The 1-based index of the parameter
- Returns** An IRowSet instance
- Throws** OdaException if a data source error occurs

ICallStatement.setNewRowSet

- Syntax** `public com.actuate.oda.IRowSet setNewRowSet(java.lang.String
parameterName) throws OdaException`

An optional method that returns an instance of IRowSet that represents the specified table input parameter. Implement this method when the driver supports named complex input parameters. The client calls IRowSet setter methods to populate the input parameter. For example:

```
IRowSet myTable = myStatement.setNewRowSet("MyStructureName");
myTable.add();
myTable.setString( 1, "myValue1" );
myTable.add();
myTable.setString( 1, "myValue2" );
```


Parameter	parameterName The name of the parameter
Returns	An IRowSet instance
Throws	OdaException if a data source error occurs
Syntax	<pre>public com.actuate.oda.IRowSet setNewRowSet(int parameterId) throws OdaException</pre> An optional method that returns an instance of IRowSet that represents the specified table input parameter. Implement this method when the driver supports complex input parameters. The client calls IRowSet setter methods to populate the input parameter. For example: <pre>IRowSet myTable = myStatement.setNewRowSet(1); myTable.add(); myTable.setString(1, "myValue1"); myTable.add(); myTable.setString(1, "myValue2");</pre>
Parameter	parameterId The 1-based index of the parameter
Returns	An IRowSet instance
Throws	OdaException if a data source error occurs

ICallStatement.setSortSpec

Syntax	<pre>public void setSortSpec(java.lang.String resultSetName, SortSpec sortBy) throws OdaException</pre> Associates a SortSpec instance with the specified columns to indicate how to sort the result set. Calling setSortSpec() with a null argument clears the previously associated SortSpec. If no sort specification is set, the rows of a result set have an implementation-specific ordering, and the getSortSpec() method returns null.
Parameters	resultSetNames The name of the result set sortBy The SortSpec instance to associate with the specified result set columns
Throws	OdaException if the sort specification is not valid or not supported by the driver

ICallStatement.wasNull

Syntax	<pre>public boolean wasNull() throws OdaException</pre>
---------------	---

ICallStatement.wasNull

Indicates whether the value of the last output parameter a getter method read is null. Implement this method when the driver supports output parameters. In each getter method, set any data structures to test using `wasNull()`.

Returns True if the last call to a getter method is null

Throws `OdaException` if a data source error occurs

Interface IClob

Syntax public interface IClob

Description IClob is an interface for processing a character large object (CLOB) data value. You implement this interface only if your ODA driver supports the CLOB data type. The methods in this interface require getting a handle for the ICLOB object. If an ODA result set column is a CLOB, use `IRResultSet.getClob()` to obtain a handle for the ICLOB object. If an ODA output parameter is a CLOB, use `ICallStatement.getClob()` to obtain a handle for the ICLOB object. An ODA input parameter does not support using a CLOB.

Method summary

Table 15-6 lists the methods for interface IClob.

Table 15-6 Methods for interface IClob

Method	Description
<code>getCharacterStream()</code>	Returns the entire CLOB value as a Java input stream
<code>getSubString()</code>	Returns a String that contains part of the CLOB value
<code>length()</code>	Returns the number of characters in the CLOB value

IClob.getCharacterStream

Syntax public Reader getCharacterStream() throws OdaException

Returns a handle to the value of the entire CLOB

Returns A `java.io.Reader` object that contains the CLOB value

Throws OdaException if a data source error occurs

IClob.getSubString

Syntax public String getSubString(long position, int length) throws OdaException

An optional method that returns the part of the CLOB value, starting from the position indicated by the position parameter, and continuing for the number of characters that the length parameter specifies. If the CLOB contains fewer characters after the specified position than the number that the length parameter specifies, `getSubString()` returns the CLOB value through the end of the CLOB.

IClob.length

This method has a default implementation to provide this functionality when you do not implement this method explicitly. Consider implementing this method if you can optimize processing the CLOB based on your knowledge of the type of data that it contains.

Parameters **length**

The number of characters to return. Using a negative value for length returns all characters through the end of the CLOB.

position

The position of the starting character. The first character in the CLOB is at position 1.

Returns A string that contains part of the CLOB value

Throws OdaException if a data source error occurs

IClob.length

Syntax `public long length( ) throws OdaException`

An optional method that returns the number of characters in a CLOB

Returns The number of characters in a CLOB

Throws OdaException if a data source error occurs

Interface IConnection

- Syntax** `public interface IConnection`
- Description** Defines the connection interface. This interface provides access to the rest of the open data access framework. Some `IDataSourceMetaData` methods, such as `getDataSourceObjects()`, expect or require an open connection.

Method summary

Table 15-7 lists the methods for interface `IConnection`.

Table 15-7 Methods for interface `IConnection`

Method	Description
<code>close()</code>	Closes the connection.
<code>commit()</code>	An optional method that makes permanent all changes implemented since the previous commit or roll-back.
<code>createStatement()</code>	Returns an <code>IStatement</code> instance based on the data source type.
<code>getMetaData()</code>	Returns an instance of <code>IConnectionMetaData</code> that contains information about the capabilities of this driver. Alternatively, this method returns an instance of <code>IDataSourceMetaData</code> based on the data source type.
<code>isOpened()</code>	Verifies that the connection is open.
<code>open()</code>	Establishes a connection based on specified connection properties.
<code>rollback()</code>	An optional method that undoes all changes that were made since the previous commit or rollback.

IConnection.close

- Syntax** `public void close() throws OdaException`
- A mandatory method that closes the connection
- Throws** `OdaException` if a data source error occurs
-

IConnection.commit

- Syntax** `public void commit() throws OdaException`

An optional method that makes permanent all changes that were implemented since the previous commit or roll-back. Use this method if the driver can manage transactions.

Throws OdaException if a data source error occurs

ICConnection.createStatement

Syntax `public com.actuate.oda.IStatement createStatement(java.lang.String dataSourceType) throws OdaException`

A mandatory method that returns an IStatement instance based on the data source type. The data source type is implementation-dependent.

Parameter **dataSourceType**
A String representation of the class of the data source type

Returns An IStatement object instance

Throws OdaException if a data source error occurs

ICConnection.getMetaData

Syntax `public com.actuate.oda.ICConnectionMetaData getMetaData( ) throws OdaException`

A mandatory method that returns an instance of ICConnectionMetaData that contains information about the capabilities of the driver. Can be called before opening the connection.

Returns An ICConnectionMetaData object instance

Throws OdaException if a data source error occurs

Syntax `public com.actuate.oda.IDataSourceMetaData getMetaData(java.lang.String dataSourceType) throws OdaException`

A mandatory method that returns an instance of IDataSourceMetaData based on the data source type. The data source type is implementation-dependent. Can be called before opening the connection.

Parameter **dataSourceType**
String representation of the class of the data source type

Returns An IDataSourceMetaData object instance

Throws OdaException if a data source error occurs

ICConnection.isOpened

Syntax public boolean isOpened() throws OdaException

A mandatory method that verifies that a connection has been established. Can be called before calling ICConnection.open. If isOpened() depends on objects that open() instantiates, in isOpened() verify that those objects are instantiated.

Returns True if a connection is established

Throws OdaException if a data source error occurs

ICConnection.open

Syntax public void open(java.util.Properties connProperties) throws OdaException

A mandatory method that establishes a connection based on specified connection properties

Parameter **connProperties**

The properties that are necessary to establish a connection

Throws OdaException if a data source error occurs

ICConnection.rollback

Syntax public void rollback() throws OdaException

An optional method that undoes all changes that were implemented since the previous commit or rollback. Use this method if the driver can manage transactions.

Throws OdaException if a data source error occurs

Interface IURLConnectionFactory

Syntax public interface IURLConnectionFactory

Description IURLConnectionFactory is the interface from which all connection factory implementations derive.

Method summary

Table 15-8 lists the methods for interface IURLConnectionFactory.

Table 15-8 Methods for interface IURLConnectionFactory

Method	Description
getConnection()	Returns an IURLConnection object that can establish a connection to the target system
setLogConfiguration()	Sets the logging configuration for the ODA run-time driver

URLConnectionFactory.getConnection

Syntax public com.actuate.oda.IURLConnection getConnection(java.lang.String connectionClassName) throws OdaException

A mandatory method that returns an IURLConnection object that can establish a connection to the target object. The specified connection is the fully qualified class name of the connection class.

Parameter **connectionClassName**
An optional String that represents the class name of the IURLConnection. A null or empty String returns the default IURLConnection object for the IURLConnectionFactory.

Returns An instance of IURLConnection

Throws OdaException if a data source error occurs

See also IURLConnection

URLConnectionFactory.setLogConfiguration

Syntax public void setLogConfiguration(int logLevel, java.lang.String logDirectory, java.lang.String logPrefix, java.lang.String formatterClassName) throws OdaException

Sets the logging configuration of the ODA run-time driver. An ODA driver can handle the logging configuration internally or use the ODA logging classes and interface in the com.actuate.oda.util.logging package.

- Parameters**
- formatterClassName**
The fully qualified class name of a LogFormatter implementation class
 - logDirectory**
The absolute path of the log directory
 - logLevel**
The minimum level of log records to publish
 - logPrefix**
The prefix to use in the log-file name
- Throws** OdaException if an ODA driver error occurs
- See also** LogManager.createLogger()

Interface IConnectionMetaData

Syntax public interface IConnectionMetaData

Description Comprehensive information about the connection.

The features that a connection supports depends on the driver that works with it. In addition, a driver can implement a feature that the connection does not offer. An ODA driver implements IConnectionMetaData to describe the connection and the driver to e.Report Designer Professional. For example, IConnectionMetaData describes the maximum number of connections that the driver can support and the maximum number of statements that the driver can support for a data source type. The data provider must ensure that IConnectionMetaData matches the capabilities of the driver. An IConnectionMetaData method throws OdaException if it gets information about a feature that the driver does not support.

All IConnectionMetaData methods can be called before the connection is open.

Method summary

Table 15-9 lists the methods for interface IConnectionMetaData.

Table 15-9 Methods for interface IConnectionMetaData

Method	Description
getConnection()	Returns the connection that produced this metadata object
getDriverMajorVersion()	Returns the major version number of this ODA driver
getDriverMinorVersion()	Returns the minor version number of this ODA driver
getDriverName()	Returns the name of this ODA driver.
getDriverVersion()	Returns the version number of this ODA driver as a String
getMaxConnections()	Returns the maximum number of active connections that the driver can support
getMaxStatements()	Returns the maximum number of active statements for any data source types that the driver can support for this connection
getOdaMajorVersion()	Returns the major version number of the ODA interface that this driver supports
getOdaMinorVersion()	Returns the minor version number of the ODA interface that this driver supports

IConnectionMetaData.getConnection

Syntax `public com.actuate.oda.IConnection getConnection( ) throws OdaException`
A mandatory method that returns the connection object that produced this metadata object

Returns The connection object that produced this metadata object

Throws `OdaException` if a data source error occurs

IConnectionMetaData.getDriverMajorVersion

Syntax `public int getDriverMajorVersion( )`
A mandatory method that returns this ODA driver's major version number

Returns The ODA driver's major version number

IConnectionMetaData.getDriverMinorVersion

Syntax `public int getDriverMinorVersion( )`
A mandatory method that returns this ODA driver's minor version number

Returns The ODA driver's minor version number

IConnectionMetaData.getDriverName

Syntax `public java.lang.String getDriverName( ) throws OdaException`
A mandatory method that returns the name of this ODA driver

Returns The ODA driver's name

Throws `OdaException` if a data source error occurs

IConnectionMetaData.getDriverVersion

Syntax `public java.lang.String getDriverVersion( ) throws OdaException`
A mandatory method that returns the version number of this ODA driver as a String. The String concatenates the major and minor version numbers.

Returns The ODA driver's version number

Throws OdaException if a data source error occurs

ICConnectionMetaData.getMaxConnections

Syntax public int getMaxConnections() throws OdaException

A mandatory method that returns the maximum number of active connections that the driver can support concurrently

Returns The maximum number of connections. Returns 0 for no limit or unknown limit.

Throws OdaException if a data source error occurs

ICConnectionMetaData.getMaxStatements

Syntax public int getMaxStatements() throws OdaException

A mandatory method that returns the maximum number of active statements of data source types that the driver can support for this connection

Returns The maximum number of any type of statements that can be prepared and executed concurrently. Returns 0 if there is no limit or the limit is unknown.

Throws OdaException if a data source error occurs

ICConnectionMetaData.getOdaMajorVersion

Syntax public int getOdaMajorVersion() throws OdaException

A mandatory method that returns the major version number of the ODA interface that this driver supports

Returns The major version number for the ODA interface

Throws OdaException if a data source error occurs

ICConnectionMetaData.getOdaMinorVersion

Syntax public int getOdaMinorVersion() throws OdaException

A mandatory method that returns the minor version number of the ODA interface that this driver supports

Returns The minor version number for the ODA interface

Throws OdaException if a data source error occurs

Interface IDataSourceMetaData

Syntax public interface IDataSourceMetaData

Description Comprehensive information about the data source.

Data source metadata describes the features that a data source supports. A driver can implement a feature the underlying data source does not offer. The methods in IDataSourceMetaData return information about the capabilities of a driver and a data source. For example, IDataSourceMetaData can indicate whether the data source supports getting multiple `ResultSet` objects from a single call to the method `IStatement.execute()` or whether the data source supports input or output parameters. Ensure that IDataSourceMetaData matches the capabilities of the driver. An IDataSourceMetaData method throws `OdaException` if it gets information about a feature that the driver does not support.

Some IDataSourceMetaData methods are callable before the connection is open. Other methods require an open connection. For example, `getDataSourceObjects()` requires an open connection. `supportsInParameters()` does not require an open connection. The following code shows acceptable call locations for these methods:

```
IDataSourceMetaData metadata = connection.getMetaData( ... );
metadata.supportsInParameters(); // connection is not opened
connection.open();
metadata.getDataSourceObjects( ... ); // requires an open connection
```

Field summary

Table 15-10 lists the fields for interface IDataSourceMetaData.

Table 15-10 Fields for interface IDataSourceMetaData

Field	Description
<code>public static final int sortModeColumnOrder</code>	A constant that indicates that each sorted column can have a different sort order
<code>public static final int sortModeNone</code>	A constant that indicates that dynamic sorting is not supported
<code>public static final int sortModeSingleColumn</code>	A constant that indicates that only one column can be sorted
<code>public static final int sortModeSingleOrder</code>	A constant that indicates that all sorted columns must be in the same sort order
<code>public static final int sqlStateSQL99</code>	A constant that indicates that <code>OdaException.getSQLState</code> returns a SQL99 <code>SQLSTATE</code> value

(continues)

Table 15-10 Fields for interface IDataSourceMetaData (continued)

Field	Description
public static final int sqlStateXOpen	A constant that indicates that OdaException.getSQLState returns an X/Open SQL CLI SQLSTATE value

Method summary

Table 15-11 lists the methods for interface IDataSourceMetaData.

Table 15-11 Methods for interface IDataSourceMetaData

Method	Description
getConnection()	Returns the connection object that produced this metadata object
getDataSource MajorVersion()	Returns the major version number of the data source
getDataSource MinorVersion()	Returns the minor version number of the data source
getDataSource Objects()	An optional method that returns the collection of objects in a data source catalog
getDataSource ProductName()	Returns the name of this data source product
getDataSource ProductVersion()	Returns the version number of this data source product as a String
getSortMode()	Returns the type of sorting that the data source supports
getSQLStateType()	Indicates whether the SQLSTATE that OdaException. getSQLState() returns is X/Open SQL CLI or SQL99
supportsIn Parameters()	Indicates whether the data source supports input parameters to IStatement
supportsMultiple OpenResults()	Indicates whether the data source supports returning multiple IResultSet objects simultaneously from an ICallStatement object
supportsMultiple ResultSets()	Indicates whether the data source supports getting multiple IResultSet objects from a single call to the method IStatement.execute()
supportsNamed Parameters()	Indicates whether the data source supports specified parameters of IStatement by name
supportsNamed ResultSets()	Indicates whether this data source supports getting one or more IResultSet objects by name

Table 15-11 Methods for interface IDataSourceMetaData

Method	Description
supportsOutParameters()	Indicates whether this data source supports output parameters to ICallStatement

IDataSourceMetaData.getConnection

Syntax public com.actuate.oda.IConnection getConnection() throws OdaException

A mandatory method that returns the connection object that produced this metadata object

Returns The connection object that produced this metadata object

Throws OdaException if a data source error occurs

IDataSourceMetaData.getDataSourceMajorVersion

Syntax public int getDataSourceMajorVersion() throws OdaException

A mandatory method that returns the major version number of the underlying data source

Returns The major version number of the data source

Throws OdaException if a data source error occurs

IDataSourceMetaData.getDataSourceMinorVersion

Syntax public int getDataSourceMinorVersion() throws OdaException

A mandatory method that returns the minor version number of the underlying data source

Returns The minor version number of the data source

Throws OdaException if a data source error occurs

IDataSourceMetaData.getDataSourceObjects

Syntax public com.actuate.oda.IResultSet getDataSourceObjects(java.lang.String catalog, java.lang.String schema, java.lang.String object, java.lang.String version) throws OdaException

IDataSourceMetaData.getDataSourceProductName

An optional method that returns the collection of objects in a data source catalog. Valid arguments to this method are implementation-dependent. Each driver must define the metadata for the `IRResultSet` that is returned. Using driver-specific classes and methods, a driver can extend the metadata that is returned.

Parameters	catalog
	The data source object catalog
	object
	The search pattern for the data source object name
	schema
	The search pattern for the schema or owner name of the data source object. Can be empty if it does not apply to the connected data source.
	version
	The data source object version
Returns	An <code>IRResultSet</code> instance of the data source objects
Throws	<code>OdaException</code> if a data source error occurs

IDataSourceMetaData.getDataSourceProductName

Syntax	<code>public java.lang.String getDataSourceProductName()</code> throws <code>OdaException</code>
	A mandatory method that returns the name of the data source product
Returns	The data source product's name
Throws	<code>OdaException</code> if a data source error occurs

IDataSourceMetaData.getDataSourceProductVersion

Syntax	<code>public java.lang.String getDataSourceProductVersion()</code> throws <code>OdaException</code>
	A mandatory method that returns the version number of the data source product as a <code>String</code> . The <code>String</code> concatenates the major and minor version numbers.
Returns	The version number of the data source
Throws	<code>OdaException</code> if a data source error occurs

IDataSourceMetaData.getSortMode

Syntax	<code>public int getSortMode()</code>
---------------	--

Returns the sort mode that this data source supports. The sort mode is one of the following integer constants, described in Class SortSpec:

- sortModeColumnOrder
- sortModeNone
- sortModeSingleColumn
- sortModeSingleOrder

Returns An integer that indicates which sort mode the data source supports

See also Class SortSpec

IDataSourceMetaData.getSQLStateType

Syntax public int getSQLStateType() throws OdaException

A mandatory method that indicates whether the SQLSTATE that OdaException.getSQLState() returns is X/Open SQL CLI or SQL99

Returns The type of SQLSTATE, which is either sqlStateXOpen or sqlStateSQL99

Throws OdaException if a data source error occurs

IDataSourceMetaData.supportsInParameters

Syntax public boolean supportsInParameters() throws OdaException

A mandatory method that indicates whether the data source supports passing input parameters to IStatement

Returns True if the data source supports input parameters

Throws OdaException if a data source error occurs

IDataSourceMetaData.supportsMultipleOpenResults

Syntax public boolean supportsMultipleOpenResults() throws OdaException

A mandatory method that indicates whether the data source supports returning multiple IResultSet objects simultaneously from an ICallStatement object

Returns True if an ICallStatement object can return multiple IResultSet objects simultaneously

Throws OdaException if a data source error occurs

IDataSourceMetaData.supportsMultipleResultSets

- Syntax** `public boolean supportsMultipleResultSets( ) throws OdaException`
A mandatory method that indicates whether the data source supports getting multiple `IRResultSet` objects from a single call to `IStatement.execute( )`
- Returns** True if a single call to `IStatement.execute( )` can get multiple `IRResultSet` objects
- Throws** `OdaException` if a data source error occurs

IDataSourceMetaData.supportsNamedParameters

- Syntax** `public boolean supportsNamedParameters( ) throws OdaException`
A mandatory method that indicates whether the data source supports passing named parameters to `IStatement`
- Returns** True if the data source supports named parameters
- Throws** `OdaException` if a data source error occurs

IDataSourceMetaData.supportsNamedResultSets

- Syntax** `public boolean supportsNamedResultSets( ) throws OdaException`
A mandatory method that indicates whether the data source supports getting one or more `IRResultSet` objects by name
- Returns** True if the data source supports getting one or more `IRResultSet` objects by name
- Throws** `OdaException` if a data source error occurs

IDataSourceMetaData.supportsOutParameters

- Syntax** `public boolean supportsOutParameters( ) throws OdaException`
A mandatory method that indicates whether the data source supports accepting output parameters from `ICallStatement`
- Returns** True if the data source supports output parameters
- Throws** `OdaException` if a data source error occurs

Interface IPParameterMetaData

- Syntax** public interface IPParameterMetaData
- Description** IPParameterMetaData is an optional interface that represents metadata for parameters of a prepared statement. All indexes in this interface are 1-based.

Field summary

Table 15-12 lists the fields for interface IPParameterMetaData.

Table 15-12 Fields for interface IPParameterMetaData

Fields	Description
public static final int parameterModeIn	A constant that indicates that the parameter is an input parameter
public static final int parameterModeInOut	A constant that indicates that the parameter is both input and output
public static final int parameterModeOut	A constant that indicates that the parameter is an output parameter
public static final int parameterModeUnknown	A constant that indicates that the input or output mode of the parameter is unknown
public static final int parameterNoNulls	A constant that indicates that the parameter does not support null values
public static final int parameterNullable	A constant that indicates that the parameter supports null values
public static final int parameterNullableUnknown	A constant that indicates that the nullability of the parameter is unknown

Method summary

Table 15-13 lists the methods for interface IPParameterMetaData.

Table 15-13 Methods for interface IPParameterMetaData

Method	Description
getParameterCount()	Returns the number of parameters in the prepared statement object for which this metadata object contains information
getParameterMode()	Returns the input or output mode of the specified parameter

(continues)

Table 15-13 Methods for interface IParameterMetaData (continued)

Method	Description
getParameterType()	Returns the java.sql.Types type of the specified parameter
getParameterTypeName()	Returns the data source-specific type name for the specified parameter
getPrecision()	An optional method that returns the maximum number of digits for the specified parameter
getScale()	An optional method that returns the maximum number of digits that can appear to the right of the decimal point for the specified parameter
isNullable()	An optional method that indicates whether the data source supports null values for the specified parameter

IParameterMetaData.getParameterCount

Syntax public int getParameterCount() throws OdaException

A mandatory method that returns the number of parameters in the prepared statement object for which the IParameterMetaData object contains information

Returns The number of parameters

Throws OdaException if a data source error occurs

IParameterMetaData.getParameterMode

Syntax public int getParameterMode(int param) throws OdaException

A mandatory method that returns the input and output mode of the specified parameter. The fields in this interface define the available return values.

Parameter **param**
The 1-based index of the parameter

Returns The input and output mode of the parameter. Valid values are:

- parameterModeUnknown
- parameterModeIn
- parameterModeInOut
- parameterModeOut

Throws OdaException if a data source error occurs

IPParameterMetaData.getParameterType

Syntax public int getParameterType(int param) throws OdaException

A mandatory method that returns the java.sql.Types type of the specified parameter. All java.sql.Types data types, such as ARRAY, INTEGER, and STRUCT, are valid return values. An ODA provider does not necessarily return all types.

Parameter **param**
The 1-based index of the parameter

Returns The java.sql.Types type of the parameter

Throws OdaException if a data source error occurs

IPParameterMetaData.getParameterTypeName

Syntax public java.lang.String getParameterTypeName(int param) throws OdaException

A mandatory method that returns the data source type name for the specified parameter type

Parameter **param**
The 1-based index of the parameter

Returns The name of the parameter type

Throws OdaException if a data source error occurs

IPParameterMetaData.getPrecision

Syntax public int getPrecision(int param) throws OdaException

An optional method that returns the maximum number of digits for the specified parameter. getPrecision() typically applies only to numeric data types. The ODA data source determines which java.sql.Types apply. The maximum precision supported on a data type can vary depending on the data source.

Parameter **param**
The 1-based index of the parameter

Returns The precision of the parameter or -1 if precision is not applicable

Throws OdaException if a data source error occurs

IPParameterMetaData.getScale

Syntax `public int getScale(int param) throws OdaException`

An optional method that returns the maximum number of digits that can appear to the right of the decimal point for the specified parameter. `getScale()` typically applies only to numeric data types. The ODA data source determines which `java.sql.Types` apply. The data source determines the maximum scale for a data type.

Parameter **param**
The 1-based index of the parameter

Returns The scale of the parameter or -1 if scale does not apply

Throws `OdaException` if a data source error occurs

IPParameterMetaData.isNullable

Syntax `public int isNullable(int param) throws OdaException`

An optional method that indicates whether the specified parameter supports null values. The fields in this interface define the available return values.

Parameter **param**
The 1-based index of the parameter

Returns The nullability of the parameter. Valid values are:

- `parameterNullableUnknown`
- `parameterNoNulls`
- `parameterNullable`

Throws `OdaException` if a data source error occurs

Interface ResultSet

Syntax public interface ResultSet

Description ResultSet is the interface for all result sets. An ResultSet object maintains a cursor that points to its current data row. Initially, the cursor is before the first row. The next() method moves the cursor to the next row until there are no more rows or until it reaches the maximum number of rows that can be returned.

The ODA host typically sets the MaxRows property for the result set to zero to retrieve the entire result set. To ensure that the ODA driver returns the correct result set, you must test explicitly for a MaxRows size of zero in next().

Method summary

Table 15-14 lists the methods for interface ResultSet.

Table 15-14 Methods for interface ResultSet

Method	Description
close()	Closes the cursor for the current ResultSet.
findColumn()	An optional method that returns the column index of the specified column name.
getBigDecimal()	Returns the decimal value of the specified column in the current row.
getBlob()	Gets the value of the specified column in the current row as an IBlob.
getClob()	Gets the value of the specified column in the current row as an IClob.
getDate()	Gets the value of the specified column in the current row as a java.sql.Date.
getDouble()	Gets the value of the specified column in the current row as a double.
getInt()	Gets the value of the specified column in the current row as an int.
getMetaData()	Returns the metadata associated with the current ResultSet.
getRow()	An optional method that returns the 1-based index position of the current row.
getString()	Gets the value of the specified column in the current row as a String.

(continues)

Table 15-14 Methods for interface ResultSet (continued)

Method	Description
getTime()	Gets the value of the specified column in the current row as a java.sql.Time.
getTimestamp()	Gets the value of the specified column in the current row as a java.sql.Timestamp.
next()	Moves the cursor down one row from its current position.
setFetchSize()	Deprecated. Use setMaxRows().
setMaxRows()	Specifies the maximum number of rows that can be fetched from this result set.
wasNull()	Checks whether the value retrieved from the previous getter method was invalid or null. Call immediately after the getter method.

ResultSet.close

Syntax public void close() throws OdaException

A mandatory method that closes the cursor associated with the current ResultSet

Throws OdaException if a data source error occurs

ResultSet.findColumn

Syntax public int findColumn(java.lang.String columnName) throws OdaException

An optional method that returns the column index of the named column. Implement this method when the driver supports access to result set columns by name.

Parameter **columnName**
The name of the column

Returns The 1-based column index

Throws OdaException if a data source error occurs

ResultSet.getBigDecimal

Syntax public BigDecimal getBigDecimal(int index) throws OdaException

Returns the decimal value of the column that is identified by its position in the current row

Parameter **index**
The position of the column

Returns The decimal value

Throws OdaException if a data source error occurs

Syntax public BigDecimal getBigDecimal(java.lang.String columnName) throws OdaException

Returns the decimal value of the named column in the current row

Parameter **columnName**
Name of the column

Returns The decimal value

Throws OdaException if a data source error occurs

ResultSet.getBlob

Syntax public IBlob getBlob(int index) throws OdaException

Returns the BLOB value in the current row for a column, using the column position to identify the desired column. After using ResultSet.getBlob(), an ODA driver must keep the IBLOB object and the BLOB's data accessible until the driver closes the result set.

Parameter **index**
The position of the column

Returns The BLOB value. Returns null if the specific column has a null value.

Throws OdaException if a data source error occurs

Syntax public IBlob getBlob(java.lang.String columnName) throws OdaException

Returns the BLOB value in the current row for a column, using the column name to identify the desired column. After using ResultSet.getBlob(), an ODA driver must keep the IBLOB object and the BLOB's data accessible until the driver closes the result set.

Parameter **columnName**
The column name

Returns The BLOB value. Returns null if the specific column has a null value.

Throws OdaException if a data source error occurs

IRResultSet.getClob

Syntax `public IClob getIClob(int index) throws OdaException`

Returns a CLOB value in the current row, using the column position to identify the column. After using `IRResultSet.getClob()`, an ODA driver must keep the ICLOB object and the CLOB's data accessible until the driver closes the result set.

Parameter **index**
The position of the column

Returns The CLOB value. Returns null if the specific column has a null value.

Throws `OdaException` if a data source error occurs

Syntax `public IClob getIClob(java.lang.String columnName) throws OdaException`

Returns a CLOB value for a named column in the current row. After using `IRResultSet.getClob()`, an ODA driver must keep the ICLOB object and the CLOB's data accessible until the driver closes the result set.

Parameter **columnName**
The column name

Returns The CLOB value. Returns null if the specific column has a null value.

Throws `OdaException` if a data source error occurs

IRResultSet.getDate

Syntax `public java.sql.Date getDate(int index) throws OdaException`

Gets the value of the column in the current row as a `java.sql.Date`. Mandatory if the driver supports the Date data type.

Parameter **index**
The 1-based column index

Returns The `java.sql.Date` value of the specified column of the current row

Throws `OdaException` if a data source error occurs

Syntax `public Date getDate(String columnName) throws OdaException`

Gets a `java.sql.Date` value for the named column in the current row. Mandatory if the driver supports the Date data type and accessing result set columns by name.

Parameter **columnName**
The column name

Returns The `java.sql.Date` value of the specified column of the current row

Throws `OdaException` if a data source error occurs

ResultSet.getDouble

Syntax `public double getDouble(int index) throws OdaException`

Gets the value of the specified column in the current row as a double. Mandatory if the driver supports the double data type.

Parameter **index**
The 1-based column index

Returns The double value of the specified column of the current row

Throws OdaException if a data source error occurs

Syntax `public double getDouble(String columnName) throws OdaException`

Gets the value of the specified column in the current row as a double. Mandatory if the driver supports the double data type and accessing result set columns by name.

Parameter **columnName**
The column name

Returns The double value of the specified column of the current row

Throws OdaException if a data source error occurs

ResultSet.getInt

Syntax `public int getInt(int index) throws OdaException`

Gets the value of the specified column in the current row as an int. Mandatory if the driver supports the integer data type.

Parameter **index**
The 1-based column index

Returns The integer value of the specified column of the current row

Throws OdaException if a data source error occurs

Syntax `public int getInt(String columnName) throws OdaException`

Gets the value of the specified column in the current row as an int. Mandatory if the driver supports the integer data type and accessing result set columns by name.

Parameter **columnName**
The column name

Returns The integer value of the specified column of the current row

Throws OdaException if a data source error occurs

IResultSet.getMetaData

Syntax `public com.actuate.oda.IResultSetMetaData getMetaData( )`
throws `OdaException`

A mandatory method that returns the metadata associated with the current `ResultSet`

Returns The metadata for the `ResultSet`

Throws OdaException if a data source error occurs

ICollection.GetRow

Syntax public int getRow() throws OdaException

An optional method that returns the current row's 1-based index position

Returns The 1-based index position of the current row

Throws OdaException if a data source error occurs

ResultSet.getString

Syntax `public java.lang.String getString(int index) throws OdaException`

Gets the value of the specified column in the current row as a String. Mandatory if the driver supports the String data type. getString() can provide the String value of a non-String column. The format of the string is implementation-dependent.

Parameter	index
	The 1-based column index

Returns The string value of the specified column of the current row

Throws OdaException if a data source error occurs

Syntax `public String getString(String columnName) throws OdaException`

Gets the value of the named column in the current row as a String. Mandatory if the driver supports the String data type and accessing result set columns by name. `getString()` can provide the String value of a non-String column. The format of the string is implementation-dependent.

Parameter	columnName
	The column name

Returns The string value of the specified column of the current row

Throws OdaException if a data source error occurs

ResultSet.getTime

Syntax `public java.sql.Time getTime(int index) throws OdaException`
 Gets the value of the specified column in the current row as a `java.sql.Time`.
 Mandatory if the driver supports the Time data type.

Parameter **index**
 The 1-based column index

Returns The `java.sql.Time` value of the specified column of the current row

Throws `OdaException` if a data source error occurs

Syntax `public Time getTime(String columnName) throws OdaException`
 Gets the value of the named column in the current row as a `java.sql.Time`.
 Mandatory if the driver supports the Time data type and accessing result set columns by name.

Parameter **columnName**
 The column name

Returns The `java.sql.Time` value of the specified column of the current row

Throws `OdaException` if a data source error occurs

ResultSet.getTimestamp

Syntax `public java.sql.Timestamp getTimestamp(int index) throws OdaException`
 Gets the value of the specified column in the current row as a `java.sql.Timestamp`.
 Mandatory if the driver supports the Timestamp data type.

Parameter **index**
 The 1-based column index

Returns The `java.sql.Timestamp` value of the specified column of the current row

Throws `OdaException` if a data source error occurs

Syntax `public Timestamp getTimestamp(String columnName) throws OdaException`
 Gets the value of the named column in the current row as a `java.sql.Timestamp`.
 Mandatory if the driver supports the Timestamp data type and accessing result set columns by name.

Parameter **columnName**
 The column name

Returns The `java.sql.Timestamp` value of the specified column of the current row

Throws `OdaException` if a data source error occurs

IResultSet.next

Syntax `public boolean next( ) throws OdaException`

A mandatory method that moves the cursor down one row from its current position

Returns True if there is a subsequent data row and the maximum rows limit has not been reached

Throws `OdaException` if a data source error occurs

Example

```
public boolean next( ) throws OdaException
{
    if( m_maxRows != 0 && m_rowsReturned >= m_maxRows )
        return false;
    // else process the next row
}
```

IResultSet.setMaxRows

Syntax `public void setMaxRows(int max) throws OdaException`

Specifies the maximum number of rows to fetch from this result set. This value should not be greater than the maximum number of rows specified in the related `IStatement`. A value of zero means that there is no limit.

Parameter **max**
The maximum number of rows to fetch from this `IResultSet`

Throws `OdaException` if a data source error occurs

IResultSet.wasNull

Syntax `public boolean wasNull( ) throws OdaException`

A mandatory method that checks whether the value of the previous getter method is invalid or null. Call immediately after the getter method. In each getter method, set any data structures used by `wasNull( )`.

Returns True if the previous getter method call was invalid or returned null

Throws `OdaException` if a data source error occurs

Interface ResultSetMetaData

Syntax public interface ResultSetMetaData

Description ResultSetMetaData is the metadata interface for all metadata implementations. ResultSetMetaData represents a row that contains metadata for each column that corresponds to the result set. All indexes in this interface are 1-based.

Field summary

Table 15-15 lists the fields for interface ResultSetMetaData.

Table 15-15 Fields for interface ResultSetMetaData

Field	Description
public static final int columnNoNulls	A constant that indicates that a column does not support null values
public static final int columnNullable	A constant that indicates that a column supports null values.
public static final int columnNullableUnknown	A constant that indicates that the nullability of the values for a column is unknown.

Method summary

Table 15-16 lists the methods for interface ResultSetMetaData.

Table 15-16 Methods for interface ResultSetMetaData

Method	Description
getColumnCount()	Returns the number of columns in the corresponding ResultSet object
getColumnDisplayLength()	Returns the display length of a specified column
getColumnLabel()	Returns the suggested title of a specified column for use in the column heading and data row variable name
getColumnName()	Returns the name of a specified column
getColumnType()	Returns the java.sql.Types type of a specified column.
getColumnTypeName()	Returns the type name of a specified column
getPrecision()	An optional method that returns the maximum number of decimal digits in a specified column

(continues)

Table 15-16 Methods for interface ResultSetMetaData (continued)

Method	Description
getScale()	An optional method that returns the maximum number of digits that can appear to the right of the decimal point in a specified column
isNullable()	An optional method that indicates the nullability of values in a specified column

ResultSetMetaData.getColumnCount

- Syntax** public int getColumnCount() throws OdaException
- A mandatory method that returns the number of columns in the corresponding ResultSet object
- Returns** The number of columns
- Throws** OdaException if a data source error occurs

ResultSetMetaData.getColumnDisplayLength

- Syntax** public int getColumnDisplayLength(int index) throws OdaException
- A mandatory method that returns the display length of the specified column
- Parameter** **index**
The 1-based column index
- Returns** The column's display length
- Throws** OdaException if a data source error occurs

ResultSetMetaData.getColumnLabel

- Syntax** public java.lang.String getColumnLabel(int index) throws OdaException
- A mandatory method that returns the suggested title of a specified column for use in the column heading and data row variable name
- Parameter** **index**
The 1-based column index
- Returns** The recommended column title
- Throws** OdaException if a data source error occurs

ResultSetMetaData.getColumnName

Syntax `public java.lang.String getColumnName(int index) throws OdaException`

A mandatory method that returns the name of the specified column

Parameter **index**
The 1-based column index

Returns The column name

Throws OdaException if a data source error occurs

ResultSetMetaData.getColumnType

Syntax `public int getColumnType(int index) throws OdaException`

A mandatory method that returns the type of the specified column. All java.sql.Types data types, such as ARRAY, INTEGER, and STRUCT, are valid return values. An ODA driver or data source does not need to return every type.

Parameter **index**
The 1-based column index

Returns The java.sql.Types type of the column

Throws OdaException if a data source error occurs

ResultSetMetaData.getColumnTypeName

Syntax `public java.lang.String getColumnTypeName(int index) throws OdaException`

A mandatory method that returns the type name of the specified column

Parameter **index**
The 1-based column index

Returns The column type name

Throws OdaException if a data source error occurs

ResultSetMetaData.getPrecision

Syntax `public int getPrecision(int index) throws OdaException`

An optional method that returns the maximum number of decimal digits of the specified column. getPrecision() typically applies only to numeric data types. The

driver determines which java.sql.Types apply. The maximum precision supported on a data type can vary depending on the data source.

- Parameter** **index**
The 1-based column index
- Returns** The column precision. Returns -1 if precision does not apply.
- Throws** OdaException if a data source error occurs

ResultSetMetaData.getScale

Syntax public int getScale(int index) throws OdaException

An optional method that returns the maximum number of digits that can appear to the right of the decimal point for the specified column. getScale() typically applies only to numeric data types. The ODA data source determines which java.sql.Types apply and sets the maximum scale for a data type.

- Parameter** **index**
The 1-based column index
- Returns** The column scale or -1 if scale does not apply
- Throws** OdaException if a data source error occurs

ResultSetMetaData.isNullable

Syntax public int isNullable(int index) throws OdaException

An optional method that indicates the nullability of values in the specified column. The fields in this interface define the available return values.

- Parameter** **index**
The 1-based column index
- Returns** The nullability status of the specified column. Valid values are:
- columnNoNulls
 - columnNullable
 - columnNullableUnknown
- Throws** OdaException if a data source error occurs

Interface IRowSet

Syntax public interface IRowSet extends IResultSet

Description IRowSet is an interface for representing a complex structure or table. All complex structures or tables implement this interface. For example, implement IRowSet to represent complex parameter data values at run time. A collection of one row can be used as a structure.

This interface is required only if the driver supports complex input or output parameters. A complex parameter's metadata can be obtained from its inherited `getMetaData()` method. An implementation of IRowSet must implement the mandatory methods that it inherits from IResultSet.

Method summary

Table 15-17 lists the methods for interface IRowSet.

Table 15-17 Methods for interface IRowSet

Method	Description
<code>absolute()</code>	Moves the cursor to the specified row
<code>add()</code>	Appends a new row to the end of this collection, and moves the cursor to the new row's position
<code>clear()</code>	An optional method that removes all the elements from this collection
<code>isEmpty()</code>	Determines whether this collection contains any elements
<code>previous()</code>	An optional method that moves the cursor up one element from its current position
<code>setBigDecimal()</code>	Sets the decimal value at the designated column
<code>setDate()</code>	Sets the date value at the specified column
<code>setDouble()</code>	Sets the double value at the specified column
<code>setInt()</code>	Sets the integer value at the specified column
<code>setString()</code>	Sets the string value at the specified column
<code>setTime()</code>	Sets the time value at the specified column
<code>setTimestamp()</code>	Sets the time stamp value at the specified column
<code>size()</code>	Returns the number of elements in this collection

Methods inherited from interface com.actuate.oda.IResultSet

close, findColumn, getBigDecimal, getBigDecimal, getDate, getDate, getdouble, getDouble, getInt, getInt, getMetaData, getRow, getString, getString, getTime, getTime, getTimestamp, getTimestamp, next, setMaxRows, wasNull

IRowSet.absolute

Syntax public boolean absolute(int rowIndex) throws OdaException

A required method that moves the cursor to the specified row

Parameter **rowIndex**
The 1-based row number

Returns True if the cursor moves successfully to the row

Throws OdaException if a data source error occurs

IRowSet.add

Syntax public int add() throws OdaException

Appends a new row to the end of this collection and moves the cursor to the position of the new row. Mandatory only for input parameters.

Returns Zero if add() fails to add a new row. Otherwise, returns the 1-based row index of the new row.

Throws OdaException if a data source error occurs

IRowSet.clear

Syntax public void clear() throws OdaException

An optional method that removes all elements from this collection

Throws OdaException if a data source error occurs

IRowSet.isEmpty

Syntax public boolean isEmpty() throws OdaException

A mandatory method that determines whether this collection contains elements

Returns True if the collection is empty

Throws OdaException if a data source error occurs

IRowSet.previous

Syntax public boolean previous() throws OdaException

An optional method that moves the cursor up one element from its current position

Returns True if the cursor moves to a valid row

Throws OdaException if a data source error occurs

IRowSet.setBigDecimal

Syntax public void setBigDecimal(int columnIndex, BigDecimal value)
throws OdaException

Sets the decimal value of the column that is identified by its position in the current row

Parameter **columnIndex**
The 1-based index position of the column

value
The decimal value

Throws OdaException if a data source error occurs

Syntax public void setBigDecimal(java.lang.String columnName, BigDecimal value)
throws OdaException

Sets the decimal value of the named column in the current row

Parameters **columnName**
Name of the column

value
The decimal value

Throws OdaException if a data source error occurs

IRowSet.setDate

Syntax public void setDate(int columnIndex, java.sql.Date value) throws OdaException

Sets the date value at the specified column. Mandatory only for input parameters. Implement this method when the driver supports the Date data type.

IResultSet.setDouble

Parameters	columnIndex The 1-based index of the column
	value The java.sql.Date value
Throws	OdaException if a data source error occurs
Syntax	public void setDate(java.lang.String columnName, java.sql.Date value) throws OdaException
	Sets the date value of the named column. Mandatory only for input parameters that support setting columns by name. Implement this method when the driver supports the Date data type.
Parameters	columnName The name of the column
	value The java.sql.Date value
Throws	OdaException if a data source error occurs

IResultSet.setDouble

Syntax	public void setDouble(int columnIndex, double value) throws OdaException
	Sets the double value at the specified column. Mandatory only for input parameters. Implement this method when the driver supports the double data type.
Parameters	columnIndex The 1-based index of the column
	value The double value
Throws	OdaException if a data source error occurs
Syntax	public void setDouble(java.lang.String columnName, double value) throws OdaException
	Sets the double value of the named column. Mandatory only for input parameters that support setting columns by name. Implement this method when the driver supports the double data type.
Parameters	columnName The name of the column
	value The double value
Throws	OdaException if a data source error occurs

Sets the integer value at the specified column. Mandatory only for input parameters. Implement this method when the driver supports the integer data type.

Parameters columnIndex

The 1-based index of the column

value

The integer value

Throws OdaException if a data source error occurs

Syntax `public void setInt(java.lang.String columnName, int value)`
 throws `OdaException`

Sets the integer value of the named column. Mandatory only for input parameters that support setting columns by name. Implement this method when the driver supports the integer data type.

Parameters **columnName**

The name of the column

value

The integer value

Throws OdaException if a data source error occurs

IRowSet.setString

Syntax `public void setString(int columnIndex, java.lang.String value)`
throws `OdaException`

Sets the string value at the specified column. Mandatory only for input parameters. Implement this method when the driver supports the String data type. Not all ODA drivers support `setString()` on a non-String type column. The format of the string parameter is implementation-dependent.

Parameters `columnIndex`

The 1-based index of the column

value

The string value

Throws OdaException if a data source error occurs

Syntax public void setString(java.lang.String columnName, java.lang.String value)
 throws OdaException

Sets the string value of the named column. Mandatory only for input parameters that support setting columns by name. Implement this method when the driver supports the String data type. Not all ODA drivers support setString() on a non-String type column. The format of the string parameter is implementation-dependent.

Parameters **columnName**
 The name of the column

value
 The string value

Throws OdaException if a data source error occurs

IRowSet.setTime

Syntax public void setTime(int columnIndex, java.sql.Time value)
 throws OdaException

Sets the time value at the specified column. Mandatory only for input parameters. Implement this method when the driver supports the Time data type.

Parameters **columnIndex**
 The 1-based index of the column

value
 The java.sql.Time value

Throws OdaException if a data source error occurs

Syntax public void setTime(java.lang.String columnName, java.sql.Time value)
 throws OdaException

Sets the time value of the named column. Mandatory only for input parameters that support setting columns by name. Implement this method when the driver supports the Time data type.

Parameters **columnName**
 The name of the column

value
 The java.sql.Time value

Throws OdaException if a data source error occurs

IRowSet.setTimestamp

Syntax `public void setTimestamp(int columnIndex, java.sql.Timestamp value)`
throws `OdaException`

Sets the time stamp value at the specified column. Mandatory only for input parameters. Implement this method when the driver supports the Timestamp data type.

Parameters	columnIndex The 1-based index of the column.
-------------------	--

value
The java.sql.Timestamp value

Throws OdaException if a data source error occurs

Syntax `public void setTimestamp(java.lang.String columnName, java.sql.Timestamp value) throws OdaException`

Sets the time stamp value of the named column. Mandatory only for input parameters that support setting columns by name. Implement this method when the driver supports the Timestamp data type.

Parameters	columnName The name of the column
-------------------	---

value
The java.sql.Timestamp value

Throws OdaException if a data source error occurs

IRowSet.size

Syntax public int size() throws OdaException

A mandatory method that returns the number of elements in this collection

Returns The number of elements in this collection.

Throws OdaException if a data source error occurs

Interface IStatement

Syntax public interface IStatement

Description IStatement is the root interface in the statement hierarchy. A statement is any string that is capable of returning data when it executes. You must always prepare IStatement before you call execute(). For example:

```
if ( statement.prepare( "SELECT * FROM TABLE" ) )
{
    // prepare succeeded
    statement.execute();
}
// else handle statement error
```

When an ODA driver or data source does not support a capability that a run-time interface exposes, the implementation for the statement setter methods should throw an `UnsupportedOperationException`. At a minimum, the message of the `UnsupportedOperationException` should contain the name of the class and method to provide the context of the exception, as shown in the following example:

```
public class MyStatementClass extends IStatement
{
    public void setInt( int parameterId, int value )
    {
        // is not supported by this Statement class
        throw new UnsupportedOperationException
            ( "MyStatementClass.setInt( int parameterId, int value )" );
    }
}
```

Method summary

Table 15-18 lists the methods for interface IStatement.

Table 15-18 Methods for interface IStatement

Method	Description
clearInParameters()	An optional method to reset all input parameters to their default values.
close()	Closes the current statement.
execute()	Executes the statement's prepared command, which can return one or more result sets.
executeQuery()	Executes the statement's prepared query and returns a single <code>ResultSet</code> object.

Table 15-18 Methods for interface IStatement

Method	Description
<code>findInParameter()</code>	An optional method that returns the 1-based index of the specified input scalar or structure parameter.
<code>getFetchSize()</code>	Deprecated. Use <code>getMaxRows()</code> .
<code>getMaxRows()</code>	Returns the maximum number of rows that can be fetched from the statement execution.
<code>getMetaData()</code>	An optional method that returns the metadata of the current result set for the current prepared IStatement. Alternatively, returns the metadata of the first result set for the current IStatement with the specified command.
<code>getMoreResults()</code>	Moves the cursor to the next result set, if one exists.
<code>getParameterMetaData()</code>	An optional method that returns the number, types, and properties of this prepared IStatement object's parameters.
<code>getParameterType()</code>	Returns the <code>java.sql.Types</code> type for a parameter.
<code>getResultSet()</code>	Returns the current result as an <code>IRResultSet</code> object.
<code>getSortSpec()</code>	Returns the sorting specification associated with this IStatement.
<code>prepare()</code>	Determines whether the command is a valid form of the implemented IStatement class.
<code>setBigDecimal()</code>	Sets the designated parameter to a decimal value.
<code>setDate()</code>	Sets the specified parameter to a Date value.
<code>setDouble()</code>	Sets the specified parameter to a double value.
<code>setFetchSize()</code>	Deprecated. Use <code>setMaxRows()</code> .
<code>setInt()</code>	Sets the specified parameter to an integer value.
<code>setMaxRows()</code>	Specifies the maximum number of rows that can be fetched from the statement execution.
<code>setProperty()</code>	An optional method that sets a named property to the specified value.
<code>setPropertyInfo()</code>	An optional method that sets the properties of this IStatement.
<code>setSortSpec()</code>	Specifies the sorting order for this IStatement.
<code>setString()</code>	Sets the specified parameter to a String value.
<code>setTime()</code>	Sets the specified parameter to a Time value.
<code>setTimestamp()</code>	Sets the specified parameter to a Timestamp value.

IStatement.clearInParameters

Syntax `public void clearInParameters( ) throws OdaException`

An optional method that resets all input parameters to their default values. If you have used other IStatement methods, such as `setInt( )`, to set a new value for an individual parameter, you can use this method to remove all changes from all input parameters.

Throws `OdaException` if a data source error occurs

IStatement.close

Syntax `public void close( ) throws OdaException`

A mandatory method that closes this statement

Throws `OdaException` if a data source error occurs

IStatement.execute

Syntax `public boolean execute( ) throws OdaException`

A mandatory method that executes the statement's prepared command. Call `execute( )` only after you call `prepare( )`. Implement this method when the driver supports getting multiple `ResultSet` objects from a single call. Implement `executeQuery( )` when the driver supports only a single result set.

Returns `True` if the command executes successfully

Throws `OdaException` if a data source error occurs

Syntax `public boolean execute(java.lang.String command) throws OdaException`

This method is equivalent to calling `prepare( command )` then `execute( )`. Because this method executes the statement immediately, `execute(String)` is available only for drivers that do not support input parameters. Implement this method when the driver supports getting multiple `ResultSet` objects from a single call. Implement `executeQuery( )` when the driver supports only a single result set.

Parameter **command**

The new command to associate with this statement

Returns `True` if the next result is a `ResultSet` object and `false` if there are no more results

Throws `OdaException` if a data source error occurs

IStatement.executeQuery

Syntax `public com.actuate.oda.IResultSet executeQuery( ) throws OdaException`

A mandatory method that executes the statement's prepared query and returns a single ResultSet object. Call this method only after you call prepare().

Returns An IResultSet instance

Throws OdaException if a data source error occurs

Syntax `public com.actuate.oda.IResultSet executeQuery(java.lang.String command) throws OdaException`

This method is equivalent to calling prepare(command) then executeQuery(). This method executes the statement's query and returns a single ResultSet object. Implement this method only for drivers that do not support input parameters.

Parameter **command**
The new query to associate with this statement

Returns An IResultSet instance

Throws OdaException if a data source error occurs

IStatement.findInParameter

Syntax `public int findInParameter(java.lang.String parameterName) throws OdaException`

An optional method that returns the 1-based index of an input parameter. Implement this method if the driver supports named input parameters.

Parameter **parameterName**
The name of the input parameter

Returns The 1-based index of the named scalar or structure input parameter

Throws OdaException if a data source error occurs

IStatement.getMaxRows

Syntax `public int getMaxRows( ) throws OdaException`

Returns the maximum number of rows that can be retrieved from each result set of this IStatement

Returns The maximum number of rows to retrieve from each result set

Throws OdaException if a data source error occurs

IStatement.getMetaData

Syntax `public com.actuate.oda.IResultSetMetaData getMetaData( )`
 throws `OdaException`

Returns the metadata of the current result set for this prepared statement. Call this method only after you call `prepare( )`. If you call this method before the `IStatement` is executed, the returned metadata refers to the first result set.

Returns An `IResultSetMetaData` instance

Throws `OdaException` if a data source error occurs

Syntax `public com.actuate.oda.IResultSetMetaData getMetaData(`
 `java.lang.String command)` throws `OdaException`

Returns the metadata of the first result set for this `IStatement` with the specified command. Calling this method is equivalent to calling `prepare( command )` then `getMetaData()`.

Parameter **command**
 The new command to associate with this `IStatement`

Returns An `IResultSetMetaData` instance

Throws `OdaException` if a data source error occurs

IStatement.getMoreResults

Syntax `public boolean getMoreResults( )` throws `OdaException`

Moves the cursor to the `IStatement`'s next result set. This method implicitly closes the current `IResultSet` object returned by the previous call to `getResultSet( )`. Implement this method when the data source supports getting multiple `IResultSet` objects from a single call to the `IStatement.execute( )` method.

Returns `True` if there are more results in this `IStatement` instance

Throws `OdaException` if a data source error occurs

IStatement.getParameterMetaData

Syntax `public com.actuate.oda.IParameterMetaData getParameterMetaData( )`
 throws `OdaException`

An optional method that returns the number, types, and properties of the current prepared `IStatement`'s parameters. Call this method only after you call `prepare( )`.

Returns An `IParameterMetaData` object describing the prepared `IStatement`'s parameters

Throws OdaException if a data source error occurs

IStatement.getParameterType

Syntax public int getParameterType(int parameterId) throws OdaException

Returns the java.sql.Types type for the specified parameter. All java.sql.Types data types, such as ARRAY, INTEGER, and STRUCT, are valid return values. An ODA driver does not necessarily return all types. Implement getParameterType() when the driver supports input or output parameters.

Parameter **parameterId**
The 1-based index of the input or output parameter

Returns The type of the input or output parameter

Throws OdaException if a data source error occurs

Syntax public int getParameterType(java.lang.String parameterName)
throws OdaException

Returns the java.sql.Types type for the named parameter. All java.sql.Types data types, such as ARRAY, INTEGER, and STRUCT, are valid return values. An ODA provider does not necessarily return all types. Implement getParameterType() when the driver supports named input or output parameters.

Parameter **parameterName**
The name of the input or output parameter

Returns The java.sql.Types type of the input or output parameter

Throws OdaException if a data source error occurs

IStatement.getResultSet

Syntax public com.actuate.oda.IResultSet getResultSet() throws OdaException

Returns the current result as an IResultSet object instance. Call this method only once for each result set and only if the data source supports getting multiple IResultSet objects from a single call to the IStatement.execute() method.

Returns An IResultSet object instance

Throws OdaException if a data source error occurs

IStatement.getSortSpec

Syntax public SortSpec getSortSpec() throws OdaException

Returns the sort specification associated with this IStatement

Returns The SortSpec associated with this IStatement or null if no SortSpec is associated

Throws OdaException if a data source error occurs

IStatement.prepare

Syntax public void prepare(java.lang.String command) throws OdaException

A mandatory method that determines whether the command is a valid form of the IStatement class. The command to do the verification cannot be empty or null.

Parameter **command**
The String to check

Throws OdaException if a data source error occurs

IStatement.setBigDecimal

Syntax public void setBigDecimal(int parameterID, BigDecimal value)
throws OdaException

Sets the designated parameter to the specified decimal value

Parameters **parameterId**
The 1-based index position of the input parameter

value
The decimal value

Throws OdaException if a data source error occurs

IStatement.setDate

Syntax public void setDate(int parameterId, java.sql.Date value) throws OdaException

Sets the specified input parameter to a Date value. Implement this method when the driver supports the Date data type and input parameters.

Parameters **parameterId**
The 1-based index of the input parameter

value
The java.sql.Date value

Throws OdaException if a data source error occurs

Syntax `public void setDate(java.lang.String parameterName, java.sql.Date value)`
 throws `OdaException`

Sets the named parameter to a `Date` value. Implement this method when the driver supports the `Date` data type and named input parameters.

Parameters **parameterName**
 The name of the input parameter

value
 The `java.sql.Date` value

Throws `OdaException` if a data source error occurs

IStatement.setDouble

Syntax `public void setDouble(int parameterId, double value)` throws `OdaException`

Sets the specified parameter to a double value. Implement this method when the driver supports the double data type and input parameters.

Parameters **parameterId**
 The 1-based index of the input parameter

value
 The double value

Throws `OdaException` if a data source error occurs

Syntax `public void setDouble(java.lang.String parameterName, double value)`
 throws `OdaException`

Sets the named parameter to a double value. Implement this method when the driver supports the double data type and named input parameters.

Parameters **parameterName**
 The name of the input parameter

value
 The double value

Throws `OdaException` if a data source error occurs

IStatement.setInt

Syntax `public void setInt(int parameterId, int value)` throws `OdaException`

Sets the specified parameter to an integer value. Implement this method when the driver supports the `Integer` data type and input parameters.

IStatement.setMaxRows

Parameters	parameterId The 1-based index of the parameter
	value An integer value
Throws	OdaException if a data source error occurs
Syntax	public void setInt(java.lang.String parameterName, int value) throws OdaException
Sets the named parameter to an integer value. Implement this method when the driver supports the integer data type and named input parameters.	
Parameters	parameterName The name of the parameter
	value An integer value
Throws	OdaException if a data source error occurs

IStatement.setMaxRows

Syntax	public void setMaxRows(int max) throws OdaException
Specifies the maximum number of rows that can be fetched in each result set of this IStatement. A value of zero means that there is no limit.	
Parameter	max The maximum number of rows to fetch
Throws	OdaException if a data source error occurs

IStatement.setProperty

Syntax	public void setProperty(java.lang.String name, java.lang.String value) throws OdaException
Sets the named property to the specified value. setProperty() supports multiple calls that use the same property name. The processing of this method is implementation-dependent. Call setProperty() before you call execute() or executeQuery().	
Parameters	name The name of the property
	value The value to set to the named property

Throws OdaException if a data source error occurs

See also IStatement.setPropertyInfo

IStatement.setPropertyInfo

Syntax public void setPropertyInfo(java.util.Properties info) throws OdaException

Set the property values for this IStatement. setPropertyInfo() supports associating a single property name with a single property value. The processing of this method is implementation-dependent. Call setPropertyInfo() before you call execute() or executeQuery().

Parameter **info**
A list of arbitrary string tag-value pairs

Throws OdaException if a data source error occurs

See also IStatement.setProperty

IStatement.setSortSpec

Syntax public void setSortSpec(SortSpec sortBy) throws OdaException

Specifies the sort specification for this IStatement. Call this method before the IStatement is executed or before getMoreResults() is called. After you call setSortSpec(), you can add additional sort keys. The final sort specification is applied to the result set or sets at execution. The ODA driver must validate the type of sort specifications that the data source supports.

Calling setSortSpec() with a null argument clears the previously associated SortSpec. If no sort specification is set, the rows of a result set have an implementation-specific order, and the getSortSpec() method returns null.

Parameter **sortBy**
The sort specification to use with this IStatement

Throws OdaException if the specified sort specification is not valid or is not supported by the driver

IStatement.setString

Syntax public void setString(int parameterId, java.lang.String value)
throws OdaException

Sets the specified parameter to the given string value. Not all open data sources or drivers support setString() on a non-String type parameter. Implement this

IStatement.setTime

method when the driver supports the String data type and input parameters. The format of the string parameter is implementation-dependent.

Parameters **parameterId**
The 1-based index of the parameter

value
A String value

Throws OdaException if a data source error occurs

Syntax public void setString(java.lang.String parameterName, java.lang.String value)
 throws OdaException

Sets the named parameter to the given string value. Not all open data sources or drivers support setString() on a non-String type parameter. Implement this method when the driver supports the String data type and named input parameters. The format of the string parameter is implementation-dependent.

Parameters **parameterName**
The name of the parameter

value
A String value

Throws OdaException if a data source error occurs

IStatement.setTime

Syntax public void setTime(int parameterId, java.sql.Time value)
 throws OdaException

Sets the specified parameter to a Time value. Implement this method when the driver supports the Time data type and input parameters.

Parameters **parameterId**
The 1-based index of the parameter

value
The java.sql.Time value

Throws OdaException if a data source error occurs

Syntax public void setTime(java.lang.String parameterName, java.sql.Time value)
 throws OdaException

Sets the named parameter to a Time value. Implement this method when the driver supports the Time data type and named input parameters.

Parameters **parameterName**
The name of the parameter

value

The java.sql.Time value

Throws OdaException if a data source error occurs

IStatement.setTimestamp

Syntax public void setTimestamp(int parameterId, java.sql.Timestamp value)
throws OdaException

Sets the specified parameter to a Timestamp value. Implement this method when the driver supports the Timestamp data type and input parameters.

Parameters **parameterId**
The 1-based index of the parameter**value**

The java.sql.Timestamp value

Throws OdaException if a data source error occurs**Syntax** public void setTimestamp(java.lang.String parameterName,
java.sql.Timestamp value) throws OdaException

Sets the named parameter to a Timestamp value. Implement this method when the driver supports the Timestamp data type and named input parameters.

Parameters **parameterName**
The name of the parameter**value**

The java.sql.Timestamp value

Throws OdaException if a data source error occurs

Class Level

Syntax

public final class Level extends Object

Description

Level indicates a log level. Use log levels to limit the types of records that are logged. When you specify a log level, you limit the records that are logged to only those at the specified log level or higher. The log level is an integer value identified by one of the following constant fields of this class, which are listed in increasing numerical value:

- ALL
- FINEST
- FINER
- FINE
- CONFIG
- INFO
- WARNING
- SEVERE
- OFF

Constructor

Syntax

public Level(java.lang.String name, int value)

Constructs a Level instance that has the specified name and log-level value

Parameters

name
The name of the Level instance

value
The log-level value

Field summary

Table 15-19 lists the fields for class Level.

Table 15-19 Fields for class Level

Field	Description
ALL	A constant that indicates the value of the ALL log level. This constant is a very large negative number to ensure it is below any other constants used as log-level values.
ALL_LEVEL	A constant that indicates the ALL log level.

Table 15-19 Fields for class Level

Field	Description
CONFIG	A constant that indicates the value of the CONFIG log level.
CONFIG_LEVEL	A constant that indicates the CONFIG log level.
FINE	A constant that indicates the value of the FINE log level.
FINE_LEVEL	A constant that indicates the FINE log level.
FINER	A constant that indicates the value of the FINER log level.
FINER_LEVEL	A constant that indicates the FINER log level.
FINEST	A constant that indicates the value of the FINEST log level.
FINEST_LEVEL	A constant that indicates the FINEST log level.
INFO	A constant that indicates the value of the INFO log level.
INFO_LEVEL	A constant that indicates the INFO log level.
OFF	A constant that indicates the value of the OFF log level. This constant is a very large positive number to ensure it is above any other constants used as log-level values.
OFF_LEVEL	A constant that indicates the OFF log level.
SEVERE	A constant that indicates the value of the SEVERE log level.
SEVERE_LEVEL	A constant that indicates the SEVERE log level.
WARNING	A constant that indicates the value of the WARNING log level.
WARNING_LEVEL	A constant that indicates the WARNING log level.

Method summary

Table 15-20 lists the methods for class Level.

Table 15-20 Methods for class Level

Method	Description
<code>equals()</code>	Returns true if the two objects have the same log-level value
<code>getName()</code>	Gets the log-level name for the current Level instance
<code>hashCode()</code>	Generates and returns a hash code based on the log-level value
<code>intValue()</code>	Gets the log-level value of the current Level instance

Methods inherited from class `java.lang.Object`

`clone`, `finalize`, `getClass`, `notify`, `notifyAll`, `toString`, `wait`, `wait`, `wait`

Level.equals

Syntax `public Boolean equals(java.lang.Object obj)`

Checks whether the current `Level` instance has the same log-level value as the `Object` provided in the argument

Parameter **obj**
The `Object` to compare

Returns `True` if the two objects have the same log-level value

Level.getName

Syntax `public java.lang.String getName( )`

Gets the log-level name for the current `Level` instance

Returns A string that contains the log-level name

Level.hashCode

Syntax `public int hashCode( )`

Generates and returns a hash code based on the log-level value

Returns The integer value of the generated hash code

Level.intValue

Syntax `public int intValue( )`

Gets the log-level value of the current `Level` instance

Returns An integer that indicates the log-level value

Class LogFormatter

- Syntax** public abstract class LogFormatter extends Object
- Description** LogFormatter is an abstract class that converts log records to formatted strings based on rules that are specified in `format()`. The default formatter class, `SimpleFormatter`, implements this class. All custom formatter classes must:
- Inherit from the LogFormatter class
 - Implement the `format()` method
 - Be included in the DriverLibraries section of the driver's configuration file, `odaconfig.xml`

Constructor

- Syntax** protected LogFormatter()
- Constructs a LogFormatter object

Method summary

Table 15-21 describes the method for class LogFormatter.

Table 15-21 Method for class LogFormatter

Method	Description
<code>format()</code>	Format the specified log record into a string

Methods inherited from class `java.lang.Object`

`clone`, `equals`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `toString`, `wait`, `wait`, `wait`

LogFormatter.format

- Syntax** public abstract String format(LogRecord record)
- Formats the specified log record as a string
- Parameter** **record**
The log record to format
- Returns** The formatted string

Example

```
public String format( LogRecord record )
{
    // <Log Level>: <Time>-<Log Message>
    String ret = record.getLevel().intValue();
    ret += ": ";
    Timestamp stamp = new Timestamp(record.getMillis() );
    ret += stamp.toString();
    ret += "- "
    ret += record.getMessage();
    ret += "\n";

    return ret;
}
```

Class Logger

- Syntax** public class Logger extends Object
- Description** Logger supports logging messages for an application. Create a logger using LogManager.createLogger(). Class Level provides fields that define the available message levels.
- See also** Class Level

Constructor

- Syntax** protected Logger(java.lang.String loggerName)
- Use LogManager.createLogger instead of this constructor to create a Logger.

Method summary

Table 15-22 lists the methods for class Logger.

Table 15-22 Methods for class Logger

Method	Description
config()	Logs a CONFIG-level message
fine()	Logs a FINE-level message
finer()	Logs a FINER-level message
finest()	Logs a FINEST-level message
getHandler()	Gets the associated Handler for this logger
getLevel()	Gets the minimum log level that is required to log a message with this Logger
getName()	Gets the name of this Logger
info()	Logs an INFO-level message
isLoggable()	Checks whether this logger can log an error or information at the specified level
log()	Logs a message at the specified level
setHandler()	Sets the Handler for this Logger
setLevel()	Sets the minimum level that is required to log messages with this Logger
severe()	Logs a SEVERE-level message for an error or exception
warning()	Logs a WARNING-level message

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Logger.config

Syntax public void config(java.lang.String message)

Logs a CONFIG-level message. Class Level provides the field that defines this message level.

Parameter **message**
The message to log

Logger.fine

Syntax public void fine(java.lang.String message)

Logs a FINE-level message. Class Level provides the field that defines this message level.

Parameter **message**
The message to log

Logger.finer

Syntax public void finer(java.lang.String message)

Logs a FINER-level message. Class Level provides the field that defines this message level.

Parameter **message**
The message to log

Logger.finest

Syntax public void finest(java.lang.String message)

Logs a FINEST-level message. Class Level provides the field that defines this message level.

Parameter **message**
The message to log

Logger.getHandler

Syntax protected Handler getHandler()

Gets the associated Handler. Class Level provides the field that defines this message level.

Returns The associated Handler

Logger.getLevel

Syntax public Level getLevel()

Gets the minimum log level that is required to log a message with this Logger. Class Level provides the fields that define the available message levels.

Returns The associated Level

Logger.getName

Syntax public java.lang.String getName()

Gets the name of the Logger

Returns The name of the Logger

Logger.info

Syntax public void info(java.lang.String message)

Logs an INFO-level message. Class Level provides the field that defines this message level.

Parameter **message**
The message to log

Logger.isLoggable

Syntax public boolean isLoggable(Level level)

Checks whether this logger can log the specified log level. If the Logger's log level is OFF, no log level is loggable, because OFF is the highest level. Class Level provides the fields that define log levels.

Returns True if the specified log level is higher or equal to the Logger's level

Logger.log

Syntax public void log(Level level, java.lang.String message)

Logs a message at the specified level. Class Level provides the field that defines this message level.

Parameters **level**
The log level of the log message

message
The message to log

Syntax public void log(Level level, java.lang.Throwable thrown)

Logs an error or exception at the specified level. Class Level provides the field that defines this message level.

Parameters **level**
The log level of the log message

throwable
The error message or exception to log

Logger.setHandler

Syntax protected void setHandler(Handler handler)

Sets the associated Handler

Returns The Handler to use with this Logger

Logger.setLevel

Syntax public void setLevel(Level level)

Sets the minimum log level that is required to log a message with this Logger. Class Level provides the fields that define the available message levels.

Returns The minimum log level required to log messages with this Logger

Logger.severe

Syntax `public void severe(java.lang.String message)`

Logs a SEVERE-level error message. Class Level provides the field that defines this message level.

Parameter **message**
The message to log

Syntax `public void severe(java.lang.Throwable thrown)`

Logs a SEVERE-level error message or exception. Class Level provides the field that defines this message level.

Parameter **thrown**
The error or exception to log

Logger.warning

Syntax `public void warning(java.lang.String message)`

Logs a WARNING-level message. Class Level provides the field that defines this message level.

Parameter **message**
The message to log

Class LoggingErrorHandler

Syntax `public class LoggingErrorHandler extends Object`

Description You can associate a LoggingErrorHandler with a Handler or an instance of a Handler subclass. The LoggingErrorHandler processes any exceptions that occur during logging. Without a LoggingErrorHandler, the log caller is not notified of errors that occur during logging. After you create a LoggingErrorHandler, use `Handler.setLoggingErrorHandler()` to associate the LoggingErrorHandler with the appropriate Handler.

Constructor

Syntax `public LoggingErrorHandler()`
Creates a LoggingErrorHandler instance

Field summary

Table 15-23 lists the fields for class LoggingErrorHandler.

Table 15-23 Fields for class LoggingErrorHandler

Field	Description
CLOSE_FAILURE	A constant that indicates failure while closing an OutputStream
FLUSH_FAILURE	A constant that indicates failure while flushing an OutputStream
FORMAT_FAILURE	A constant that indicates failure while formatting
GENERIC_FAILURE	A constant that indicates a type of failure that is not covered by the other constants
OPEN_FAILURE	A constant that indicates failure while opening an OutputStream
WRITE_FAILURE	A constant that indicates failure while writing an OutputStream

Method summary

Table 15-24 describes the method for class LoggingErrorHandler.

Table 15-24 Method for class LoggingErrorHandler

Method	Description
error()	Outputs the message, exception, and error code to System.err when a failure occurs

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

LoggingErrorHandler.error

Syntax public void error(java.lang.String message, java.lang.Exception exception, int errorCode)

Outputs the message, exception, and error code to System.err when a failure occurs. The error code is one of the following constants: CLOSE_FAILURE, FLUSH_FAILURE, FORMAT_FAILURE, GENERIC_FAILURE, OPEN_FAILURE, WRITE_FAILURE. For more information about these constants, see the field summary for this class.

Parameters **message**
The error message

errorCode
The error code constant

exception
The exception that caused the Handler to fail

Class LogManager

Syntax public static class LogManager extends Object

Description LogManager is a class to that maintains a set of Loggers. You can use LogManager to create and retrieve Loggers. You can use related classes to log information to the Loggers. To use LogManager, the ODA driver must specify the <TraceLogging> element in its configuration file, odaconfig.xml.

Method summary

Table 15-25 lists the methods for class LogManager.

Table 15-25 Methods for class LogManager

Method	Description
createLogger()	Returns a Logger that has the specified name and necessary log configuration information.
getLogFileName()	Deprecated. Examine the odaconfig.xml file for the prefix and log directory that was used to create log file names.
getLogger()	Returns a new Logger that has the specified name and log configuration information. Alternatively, returns an existing Logger that has the specified name and updated log configuration information.
getLogLevel()	Deprecated. Use Logger.getLevel().
logConfig()	Deprecated. Use Logger.config().
logException()	Deprecated. Use Logger.exception().
logFine()	Deprecated. Use Logger.fine().
logFiner()	Deprecated. Use Logger.finer().
logFinest()	Deprecated. Use Logger.finest().
logInfo()	Deprecated. Use Logger.info().
logSevere()	Deprecated. Use Logger.severe().
logWarning()	Deprecated. Use Logger.warning().
setLogConfiguration()	Deprecated. Use LogManager.getLogger().

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

LogManager.createLogger

Syntax `public Logger createLogger(java.lang.String loggerName, int logLevel, java.lang.String logDirectory, java.lang.String logPrefix, java.lang.String formatterClassName) throws IllegalArgumentException`

Creates a Logger that has the specified name and the specified log configuration information. Specify a name that is appropriate to the application using the log to avoid name collision. Each logger must have a unique name, because the LogManager manages the loggers by name. For example, use `javax.sql` as the name of a logger for a `javax.sql` package.

The format of the log file names is `<logPrefix>-YYMMDD-hhmmss.log`.

If you do not specify a `formatterClassName` or provide a null value, the default `LogFormatter` class converts log records to formatted strings. If you create a custom log formatter, the class must inherit from `com.actuate.oda.logging.LogFormatter` and implement `format()` as specified for `LogFormatter.format()`.

Parameters **formatterClassName**
The name of the logFormatter class to use

logDirectory
The directory in which to store the log files

loggerName
The name of the logger to create

logLevel
The minimum log level for messages to be logged. Class `Level` provides the fields that define the available message levels.

logPrefix
The required file-name prefix for the log-file name

Returns The created Logger instance

Throws `IllegalArgumentException` if a Logger of that name already exists

LogManager.getLogger

Syntax `public static Logger getLogger(java.lang.String loggerName)`
Gets the existing Logger with the specified name

Parameter **loggerName**
The name of the logger to create

LogManager.getLogger

Returns The Logger instance with the specified name, or null if no Logger with that name exists

Syntax `public static Logger getLogger(java.lang.String loggerName, int logLevel, java.lang.String logDirectory, java.lang.String logPrefix, java.lang.String formatterClassName)`

If no Logger has the specified name, `getLogger()` creates the Logger with the specified name and log configuration information. If a Logger with the specified name exists, `getLogger()` updates the Logger with the specified log configuration information. The Logger continues to use the same log files unless you specify a different log directory or log prefix.

Parameters **formatterClassName**
The name of the logFormatter class to use

logDirectory
The directory to store the log files

loggerName
The name of the logger to create

logLevel
The minimum log level for messages to be logged. Class Level provides the fields that define the available message levels.

logPrefix
The required file-name prefix for the log-file name

Returns The created or updated Logger instance that has the specified name

Class LogRecord

Syntax public class LogRecord extends Object implements Serializable

Description LogRecord contains information that a Logger can log

Constructor

Syntax public LogRecord(Level level, java.lang.String message)

Constructs a LogRecord instance

Parameters **level**
The level of the log message

message
The message to log

Method summary

Table 15-26 lists the methods for class LogRecord.

Table 15-26 Methods for class LogRecord

Method	Description
getLevel()	Gets the log level
getMessage()	Gets the log message
getMillis()	Gets the log time
getThrown()	Gets the associated error or exception
setLevel()	Sets the log level to the specified level
setMessage()	Sets the log message to the specified value
setMillis()	Sets the log time to the specified value
setThrown()	Sets the specified error or exception

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

LogRecord.getLevel

Syntax public Level getLevel()

LogRecord.getMessage

Gets the log level of the log record

Returns A log level instance

LogRecord.getMessage

Syntax public String getMessage()

Gets the log message of the log record

Returns A String that contains the log message

LogRecord.getMillis

Syntax public long getMillis()

Gets the log time of the log record

Returns The log time

LogRecord.getThrown

Syntax public java.lang.Throwable getThrown()

Gets the error or exception of the log record

Returns A Throwable instance for the error or exception

LogRecord.setLevel

Syntax public void setLevel(Level level)

Sets the log level for the log record

Parameter **level**

The log level. Class Level provides the fields that define the available message levels.

LogRecord.setMessage

Syntax public void setMessage(java.lang.String message)

Sets the log message for the log record

Parameter **message**
The log message

LogRecord.setMillis

Syntax `public void setMillis(long millis)`
Sets the log time of the log record

Parameter **millis**
The log time

LogRecord.setThrown

Syntax `public void setThrown(java.lang.Throwable thrown)`
Sets the error or exception for the log record

Parameter **thrown**
The Throwable instance for the error or exception

Class OdaException

Syntax public class OdaException extends Exception

Description An exception that provides information about a data source access error or other errors. An open data access exception communicates errors from the data source driver to e.Report Designer Professional or Actuate iServer. Each OdaException provides the following information:

- A string that describes the error. This string is the Java Exception message, which you can retrieve using getMessage().
- A SQLSTATE string, which follows the conventions of either XOPEN SQLSTATE or SQL 99. Use IDataSourceMetaData.getSQLStateType() to determine whether the driver returns XOPEN or SQL 99.
- An integer error code that is specific to each vendor. The underlying data source returns this error code.

OdaException inherits from java.lang.Exception.

External exceptions

Actuate's open data access framework recognizes only OdaException and RuntimeException. An open data source implementation can contain code that throws other exceptions, such as ParseException or IOException. To make these types of exceptions available to e.Report Designer Professional or Actuate iServer, complete the following tasks in this order:

- Catch the exception.
- Construct an OdaException.
- Call OdaException.initCause().
- Pass in the exception.

Localizing exceptions

OdaException provides a exception class for describing data source errors but does not provide a mechanism for localizing the messages. An ODA driver must provide its own message localization, if localization is required. The driver can pass localized messages into the OdaException instance or localize the message another way. You can create an exception subclass from OdaException to support message localization. For example, a subclass can have the following constructor:

```
public MyOdaExceptionSubClass( int errorCode, Locale locale );
```

MyOdaExceptionSubClass.getLocalizedMessage can look up the error message associated with the error number.

Unsupported operations

If an ODA driver or data source does not support a capability that a run-time interface exposes, the implementation for the statement setter methods should throw an `UnsupportedOperationException`. At a minimum, the message of the `UnsupportedOperationException` should contain the name of the class and method to provide the context of the exception, as shown in the following code:

```
public class MyStatementClass extends IStatement
{
    public void setInt( int parameterId, int value ) {
        // is not supported by this Statement class
        throw new UnsupportedOperationException
            ( "MyStatementClass.setInt( int parameterId, int value )" );
    }
}
```

Constructors

Syntax `public OdaException( )`

Constructs an `OdaException` object where the message defaults to null, `SQLSTATE` defaults to null, and `vendorCode` defaults to 0.

Syntax `public OdaException(java.lang.String message)`

Constructs an `OdaException` object with a message. The `SQLSTATE` defaults to null. The `vendorCode` defaults to 0.

Parameter **message**
A description of the exception

Syntax `public OdaException(java.lang.String message, java.lang.String sqlState)`

Constructs an `OdaException` object with a message and `SQLSTATE`. The `vendorCode` defaults to 0.

Parameters **message**
A description of the exception

sqlState
An XOPEN or SQL 99 code that identifies the exception

Syntax `public OdaException(java.lang.String message, java.lang.String sqlState, int vendorCode)`

Constructs a fully specified `OdaException` object

Parameters **message**
A description of the exception

sqlState
An XOPEN or SQL 99 code that identifies the exception

vendorCode

An exception code that the data source vendor specifies

Method summary

Table 15-27 lists the methods for class OdaException.

Table 15-27 Methods for class OdaException

Method	Description
getCause()	Returns the cause of this OdaException
getErrorCode()	Returns the vendor-specific exception code for this OdaException object
getNextException()	Returns the OdaException that is chained to this OdaException object
getSQLState()	Returns the SQLSTATE of this OdaException object
initCause()	Initializes the cause of this OdaException
setNextException()	Adds an OdaException object to the end of the chain

Methods inherited from class java.lang.Throwable

fillInStackTrace, getLocalizedMessage, getMessage, getStackTrace, printStackTrace, printStackTrace, printStackTrace, setStackTrace, toString

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

OdaException.getCause

Syntax public java.lang.Throwable getCause()

Returns the cause of this OdaException

Overrides getCause in class Throwable

Returns The cause of this OdaException or null if the cause is nonexistent or unknown

OdaException.getErrorCode

Syntax public int getErrorCode()

Returns the vendor-specified exception code for this OdaException object

Returns The vendor's error code

OdaException.getNextException

Syntax public com.actuate.oda.OdaException getNextException()

Returns the OdaException instance that is chained to this OdaException object

Returns The next OdaException object in the chain or null if there are none

OdaException.getSQLState

Syntax public java.lang.String getSQLState()

Returns the SQLSTATE of this OdaException object

Returns The SQLSTATE value

OdaException.initCause

Syntax public java.lang.Throwable initCause(java.lang.Throwable cause)
throws IllegalArgumentException, IllegalStateException

Initializes the cause of this OdaException to the specified value. Call this method only once.

Overrides initCause in class Throwable

Parameter **cause**

The cause of this OdaException. A null value indicates that the cause is nonexistent or unknown.

Returns A reference to this OdaException

Throws java.lang.IllegalArgumentException if the cause is this OdaException, or
IllegalStateException if this method has already been called on this OdaException

OdaException.setNextException

Syntax public void setNextException(com.actuate.oda.OdaException nextException)

Adds an OdaException object to the end of the OdaException chain

Parameter **nextException**

The new OdaException object to add to the OdaException chain

Class SimpleFormatter

Syntax public class SimpleFormatter extends LogFormatter

Description Formats the information provided in the LogRecord

Constructor

Syntax public SimpleFormatter()

Constructs a SimpleFormatter instance

Method summary

Table 15-28 describes the method for class SimpleFormatter.

Table 15-28 Method for class SimpleFormatter

Method	Description
format()	Creates a formatted String that contains the information in the LogRecord

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

SimpleFormatter.format

Syntax public java.lang.String format(LogRecord record)

Creates a formatted String that contains the information in the LogRecord

Specified by format in class LogFormatter

Parameter **record**
The LogRecord to format

Returns The formatted String

Class SortSpec

- Syntax

public class SortSpec extends Object
- Description

A class that encapsulates one or more sort keys to associate with an IStatement. This association enables an ODA host to dynamically specify how to sort one or more individual result set columns. This ability is useful for data sources that do not accept sorting specifications, such as an ORDER BY clause, in the command text. Sort modes provide information about the available sorting types. You can extend this class to provide additional ways to handle sort modes and keys.

Constructor

- Syntax

public SortSpec(int sortMode)

Constructs a SortSpec instance based on the sortMode. The sort mode can be one of the following constants, defined by Interface IDataSourceMetaData:

- sortModeColumnOrder
 - sortModeNone
 - sortModeSingleColumn
 - sortModeSingleOrder
- Parameter

sortMode

The sort mode

Field summary

Table 15-29 lists the fields for class SortSpec.

Table 15-29 Fields for class SortSpec

Fields	Description
sortAsc	A constant that indicates ascending sort order.
sortDesc	A constant that indicates descending sort order.

Method summary

Table 15-30 lists the methods for class SortSpec.

Table 15-30 Methods for class SortSpec

Method	Description
addSortKey()	Adds a sort key to specify the dynamic sort criteria

(continues)

Table 15-30 Methods for class SortSpec (continued)

Method	Description
getSortColumn()	Returns the result set column name of the sort key at the specified index position
getSortColumns()	Returns an array of all column names in the sort keys of a SortSpec instance that has a sortModeSingleOrder sort mode
getSortKeyCount()	Returns the number of sort keys associated with the SortSpec instance
getSortMode()	Returns the sort mode of this SortSpec
getSortOrder()	Returns the sort order for a specific sort key or all sort keys of a SortSpec instance
toString()	Returns a string representation of the SortSpec

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

SortSpec.addSortKey

Syntax public void addSortKey(java.lang.String columnName, int sortOrder)

Adds a sort key to specify the dynamic sort criteria for ODA providers that support one of the following sort modes:

- sortModeColumnOrder
- sortModeNone
- sortModeSingleColumn
- sortModeSingleOrder

An IllegalStateException is thrown if the specified sort key does not conform to the sortMode of the SortSpec.

The sort conditions are specified using an ordered list of one or more sort keys. Data is sorted primarily by the first sort key. Within each first sort key value, the data is sorted by the second sort key, and so on. For each sort key, specify the column name and a sort-order constant. Sort-order constants are either sortAsc or sortDesc. For more information about the sort-order constants, see the Field summary section of this class.

Parameters **columnName**
The name of the result set column to sort

sortOrder

The value that represents the sort order

- Throws**
- `IllegalArgumentException` if `columnName` is empty or if sort order is not one of the sort mode constants provided by `Interface IDataSourceMetaData`
 - `IllegalStateException` in the situations described in Table 15-31

Table 15-31 Exceptions raised when adding `sortMode`

Sort mode	Exception raised by attempting to add
<code>sortModeNone</code>	Any sort key
<code>sortModeSingleColumn</code>	A second sort key
<code>sortModeSingleOrder</code>	A sort key with a sort order that does not match the sort order of any existing sort keys

- `NullPointerException` if `columnName` is null

See also `Interface IDataSourceMetaData`

SortSpec.getSortColumn

Syntax `public java.lang.String getSortColumn(int index)`
Returns the result set column name of the sort key at the specified index position. The index starts at position one.

Parameters **index**
The index of the sort key

Returns The name of the result set column for the specified sort key

Throws `IndexOutOfBoundsException` if the index is less than one or greater than the number of sort keys

SortSpec.getSortColumns

Syntax `public java.lang.String[] getSortColumns`
Returns an array of all column names in the sort keys of a `SortSpec` instance with a `sortModeSingleOrder` sort mode

Returns An array of column names if the `SortSpec` instance has sort keys or an empty array if the `SortSpec` instance does not have sort keys

Throws `IllegalStateException` if the mode of the `Sort Spec` instance is not `sortModeSingleOrder`

SortSpec.getSortKeyCount

Syntax `public int getSortKeycount( )`

Returns the number of sort keys associated with the SortSpec instance

Returns The number of sort keys associated with the SortSpec instance

SortSpec.getSortMode

Syntax `public int getSortMode( )`

Returns the sort mode of this SortSpec. The sort mode is one of the following Interface IDataSourceMetaData field constants:

- `sortModeColumnOrder`
- `sortModeNone`
- `sortModeSingleColumn`
- `sortModeSingleOrder`

Returns Returns the sort mode of the SortSpec

See also Interface IDataSourceMetaData

SortSpec.getSortOrder

Syntax `public int getSortOrder( )`

Returns the sort order for all sort keys of a SortSpec instance with `sortModeSingleOrder` sort mode. Sort order constants are either `sortAsc` or `sortDesc`.

Returns The sort order specified for the SortSpec instance, or `sortAsc` if no sort order is specified for the SortSpec instance

Throws `IllegalStateException` if the mode of the Sort Spec instance is not `sortModeSingleOrder`

Syntax `public int getSortOrder(int index)`

Returns the sort order for the sort key at the index position. The index starts at position 1.

Parameter **index**
The index position of the sort key

Returns The sort order specified for the sort key at the index position

Throws `IndexOutOfBoundsException` if the index is less than one or greater than the number of sort keys

SortSpec.toString

Syntax `public java.lang.String toString( )`
Returns a string representation of the `SortSpec`

Returns A string representation of the `SortSpec`

Class StreamHandler

Syntax public class StreamHandler extends Handler

Description StreamHandler obtains records from a Logger and outputs them to a stream

Constructor

Syntax public StreamHandler()

Constructs a StreamHandler with no output stream

Syntax public StreamHandler(OutputStream output, LogFormatter formatter)

Constructs a StreamHandler with the specified output stream and LogFormatter

Parameters **formatter**

The LogFormatter to use to format the records

output

The output stream to use to publish records

Method summary

Table 15-32 lists methods for class StreamHandler.

Table 15-32 Methods for class StreamHandler

Method	Description
close()	Closes the current StreamHandler and frees up resources that the StreamHandler uses
finalize()	Cleans up the StreamHandler by calling StreamHandler.close()
flush()	Flushes the buffered output to the output stream
isLoggable()	Checks whether to log the specified record and whether the StreamHandler has an associated output stream
publish()	Formats and publishes the specified record
setFormatter()	Sets the LogFormatter
setOutputStream()	Sets the output stream

Methods inherited from class com.actuate.oda.util.logging.Handler

getFilter, getFormatter, getLevel, getLoggingErrorHandler, reportError, setFilter, setLevel, setLoggingErrorHandler

Methods inherited from class java.lang.Object

clone, equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

StreamHandler.close

Syntax public void close()
 Closes the current output stream

StreamHandler.finalize

Syntax public void finalize()
 Cleans up this StreamHandler instance by calling StreamHandler.close()

StreamHandler.flush

Syntax public void flush()
 Flushes the buffered output to the output stream

Specified by flush in class Handler

StreamHandler.isLoggable

Syntax public boolean isLoggable(LogRecord record)
 Checks whether to log the specified record by determining whether the record has a sufficiently high log level, the record passes the associated Filter, and the StreamHandler has an associated output stream.

Overrides isLoggable in class Handler

Parameter **record**
 The log record to check

Returns True to log the record

StreamHandler.publish

Syntax public void publish(LogRecord record)

StreamHandler.setFormatter

Publishes a record to the appropriate destination if it has a sufficiently high log level, passes any associated Filter, and the StreamHandler has an associated output stream. Publish() also formats the record, if necessary.

Specified by publish in class Handler

Parameter **record**
The log record to format and publish

StreamHandler.setFormatter

Syntax public void setFormatter(LogFormatter formatter)

Sets the formatter that will format the log records. If the formatter is null, SimpleFormatter is the default formatter.

Overrides setFormatter in class Handler

Parameters **formatter**
The formatter to set

StreamHandler.setOutputStream

Syntax public void setOutputStream(OutputStream outputStream)

Sets the output stream

Parameter **outStream**
The output stream object

Class StringSubstitutionUtil

Syntax public final class StringSubstitutionUtil extends Object

Description StringSubstitutionUtil supports locating embedded delimited strings and replacing them with different strings. Strings can be substituted by index position or by name.

Typically this class is used to identify and replace parameters. For example, if variables are identified by a preceding colon, such as :MyVariable, StringSubstitutionUtil can replace :MyVariable with the string that was defined to replace that variable name.

Method summary

Table 15-33 lists methods for class StringSubstitutionUtil.

Table 15-33 Methods for class StringSubstitutionUtil

Method	Description
getDelimitedStringCount()	Returns the number of delimited strings in the text argument
substituteByIndex()	Substitutes strings based on their index positions
substituteByName()	Substitutes strings based on their names

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

StringSubstitutionUtil.getDelimitedStringCount

Syntax public static int getDelimitedStringCount(java.lang.String text, java.lang.String startDelimiter)

Returns the number of named and unnamed delimited strings in the text argument. An unnamed delimited string consists of only the delimiters. A named delimited string has a start delimiter that is followed immediately by a name and optionally by an end delimiter. Use this syntax if your named delimited string is not marked by an end delimiter. For example, if the start delimiter is ?, ? is an unnamed delimited string, and ?parameter1 is a named delimited string.

Parameters **startDelimiter**
The string that contains the delimiter

text

The string that contains delimited strings

Returns The number of delimited strings

Throws NullPointerException if the text or startDelimiter is null

Example getDelimitedStringCount() returns 3 if the startDelimiter is:

" ; "

and the text is:

"select :param1 , :param2 from :param3"

Syntax public static int getDelimitedStringCount(java.lang.String text, java.lang.String startDelimiter, java.lang.String endDelimiter)

Returns the number of named and unnamed delimited strings in the text argument. An unnamed delimited string consists of only the delimiters. A named delimited string has a start delimiter that is followed immediately by a name and optionally by an end delimiter. Use this syntax if your named delimited strings are labeled by delimiters at the beginning and end of the string to replace. For example, if the start delimiter is ?, and the end delimiter is ;, ?: is an unnamed delimited string, and ?parameter1: is a named delimited string.

Parameters **endDelimiter**

The string that contains the delimiter that marks the end of the string to replace

startDelimiter

The string that contains the delimiter that marks the beginning of the string to replace

text

The string that contains delimited strings

Returns The number of delimited strings

Throws NullPointerException if the text, startDelimiter, or endDelimiter is null

Example getDelimitedStringCount() returns 1 if the startDelimiter is:

"<start>"

and the endDelimiter is:

"<end>"

and the text is:

"select <start>param1<end> from CUSTOMER"

Syntax public static int getDelimitedStringCount(java.lang.String text, java.lang.String startDelimiter, boolean requiresNamedDelimiters)

Returns the number of delimited strings in the text argument. An unnamed delimited string consists of only the delimiters. A named delimited string has a

start delimiter that is followed immediately by a name and optionally by an end delimiter. Use this syntax if your named delimited strings are labeled only by a start delimiter. For example, if the start delimiter is ?, ? is an unnamed delimited string, and ?parameter1 is a named delimited string.

If requiresNamedDelimiters is true, it returns the number of named delimited strings. Otherwise, it returns the number of named and unnamed delimited strings.

Parameters **startDelimiter**

The string that contains the delimiter

text

The string that contains delimited strings

Returns The number of delimited strings

Throws NullPointerException if the text or startDelimiter is null

Example getDelimitedStringCount() returns 3 if the startDelimiter is:

" : "

and the text is:

"select :param1 , :param2 from :param3"

Syntax public static int getDelimitedStringCount(java.lang.String text, java.lang.String startDelimiter, java.lang.String endDelimiter, boolean requiresNamedDelimiters)

Returns the number of delimited strings in the text argument. An unnamed delimited string consists of only the delimiters. A named delimited string has a start delimiter that is followed immediately by a name and optionally by an end delimiter. Use this syntax if your named delimited strings are labeled by delimiters at the beginning and end of the string to replace. For example, if the start delimiter is ? and the end delimiter is ;, ? is an unnamed delimited string, and ?parameter1: is a named delimited string.

If requiresNamedDelimiters is true, it returns the number of named delimited strings. Otherwise, it returns the number of named and unnamed delimited strings.

Parameters **endDelimiter**

The string that contains the delimiter that marks the end of the string to replace

startDelimiter

The string that contains the delimiter that marks the beginning of the string to replace

text

The string that contains delimited strings

Returns The number of delimited strings

Throws `NullPointerException` if the text, startDelimiter, or endDelimiter is null

Example `getDelimitedStringCount()` returns 1 if the startDelimiter is:

`"<start>"`

and the endDelimiter is:

`"<end>"`

and the text is:

`"select <start>param1<end> from CUSTOMER"`

StringSubstitutionUtil.substituteByIndex

Syntax `public static java.lang.String substituteByIndex(java.lang.String text,
java.lang.String startDelimiter, List substitutionList)`

Finds strings marked by a start delimiter, and substitutes them based on their position in the text. Each string is replaced by the corresponding string in the indexed substitution list. A startDelimiter cannot be preceded by an alphanumeric character, an underscore, or an escape character.

Parameters **startDelimiter**

The string that contains the delimiter that marks the beginning of the string to replace

substitutionList

The list of values to substitute for the delimited strings

text

The string that contains delimited strings

Returns A String that contains the text with all delimited strings substituted

Throws `NullPointerException` if the text, startDelimiter, or substitutionList is null

Example If myList contains "ID" and "NAME":

```
substituteByIndex("SELECT STUDENT.:COLUMN, STUDENT.:COLUMN FROM  
STUDENT", ":", myList)
```

returns:

```
"SELECT STUDENT.ID, STUDENT.NAME FROM STUDENT"
```

Example If myList contains "1000" and "2000":

```
substituteByIndex("SELECT ID FROM CUSTOMER WHERE ID >= :startRange  
AND ID <= :endRange", ":", myList)
```

returns:

```
"SELECT ID FROM CUSTOMER WHERE ID >= 1000 AND ID <= 2000"
```


Syntax `public static java.lang.String substituteByIndex(java.lang.String text, java.lang.String startDelimiter, java.lang.String endDelimiter, List substitutionList)`

Finds the beginning and end of strings marked by delimiters, and substitutes them based on their position in the text

Parameters **endDelimiter**
The string that contains the delimiter that marks the end of the string to replace

startDelimiter
The string that contains the delimiter that marks the beginning of the string to replace

substitutionList
The list of values to substitute for the delimited strings

text
The string that contains delimited strings

Returns A String that contains the text with all delimited strings substituted

Throws `NullPointerException` if the text, startDelimiter, endDelimiter, or substitutionList is null

Example If myList contains "STUDENT" and "NAME":

```
substituteByIndex("SELECT <start>TABLE<end>.<start>COLUMN<end>
  FROM STUDENT", "<start>", "<end>", myList)
```

returns:

```
"SELECT STUDENT.NAME FROM STUDENT"
```

Example If myList contains "1000":

```
substituteByIndex("SELECT ID FROM CUSTOMER WHERE ID =
  <start>number<end>", "<start>", "<end>", myList)
```

returns:

```
"SELECT ID FROM CUSTOMER WHERE ID = 1000"
```

StringSubstitutionUtil.substituteByName

Syntax `public static java.lang.String substituteByName(java.lang.String text, java.lang.String startDelimiter, Map nameValues)`

Finds strings marked by a start delimiter, and substitutes them based on their name

Parameters **nameValues**
The map of pairs of names and values to use in substituting strings

startDelimiter

The string that contains the delimiter that marks the beginning of the string to replace

text

The string that contains delimited strings

Returns A String that contains the text with all delimited strings substituted

Throws NullPointerException if the text, startDelimiter, or nameValues is null

Example If myMap specifies the following pairs:

```
PARAM, "PEOPLE"  
PARAM1, "NAME"
```

then:

```
substituteByName("SELECT ?PARAM.?PARAM1 FROM ?PARAM","?", myMap)
```

returns:

```
"SELECT PEOPLE.NAME FROM PEOPLE"
```

Example If myMap specifies the following pairs:

```
startRange, "1000"  
endRange, "2000"
```

then:

```
substituteByName("SELECT ID FROM CUSTOMER WHERE ID >= :startRange  
AND ID <= :endRange",":", myMap)
```

returns:

```
"SELECT ID FROM CUSTOMER WHERE ID >= 1000 AND ID <= 2000"
```

Syntax `public static java.lang.String substituteByName(java.lang.String text,
java.lang.String startDelimiter, java.lang.String endDelimiter, Map
nameValues)`

Finds the beginning and end of strings marked by delimiters, and substitutes them based on their name

Parameters **endDelimiter**

The string that contains the delimiter that marks the end of the string to replace

nameValues

The map of pairs of names and values to use in substituting strings

startDelimiter

The string that contains the delimiter that marks the beginning of the string to replace

text

The string that contains delimited strings

Returns A String that contains the text with all delimited strings substituted as specified by the map, nameValues

Throws NullPointerException if the text, startDelimiter, endDelimiter, or nameValues is null

Example If myMap specifies the following pairs:

```
PARAM, "PEOPLE"
```

```
PARAM1, "NAME"
```

then:

```
substituteByName("SELECT :PARAM::PARAM1: FROM :PARAM:", ":", ":",  
    myMap)
```

returns:

```
"SELECT PEOPLE.NAME FROM PEOPLE"
```

Example If myMap specifies the following pair:

```
number, "1000"
```

then:

```
substituteByName("SELECT ID FROM CUSTOMER WHERE ID =  
    <start>number<end> AND SID =  
    <start>number<end>", "<start>", "<end>", myMap)
```

returns:

```
"SELECT ID FROM CUSTOMER WHERE ID = 1000 AND SID <= 1000"
```

StringSubstitutionUtil.substituteByName

Index

Symbols

- , (comma) character 80
- : (colon) character 146
 - query statements and 147, 151
- ? (question mark) character 151
- " (double quotation mark) character
 - CSV text files and 80
- * (asterisk) character
 - parameters and 134
 - query statements and 130

A

- absolute method 320
- absolute paths 18
- Access databases 160
- accessing
 - Column Editor 62, 65, 137
 - custom data sources 38
 - data 4, 28, 32, 90
 - Data Row Editor 59, 136
 - Database Browser 104
 - databases 168
 - expression builder 116, 123
 - Information Object Query Builder 76
 - information objects 73
 - multiple input sources 29, 45, 50
 - multiple result sets 190–195
 - ODA configurations 206
 - ODA data sources 200, 201, 202, 236
 - ODBC data sources 90
 - Properties page 56
 - Query Editor 104
 - resources 12
 - sample configurations 15
 - sample ODA driver 200
 - Scratch Pad 56
 - Stored Procedure Data Source Builder 171
 - Stored Procedure Name Editor 173
 - stored procedures 168, 186
 - Textual Query Editor 128
- AcConnection class 99
- AcDataAdapter class 31, 32
- AcDataRowBuffer class 33
- AcDataSorter class 39
- AcDataSource class 38, 80
- AcDB2Connection class 99
- AcDBConnection class 99
- AcDBCursor class 99
- AcDBStatement class 99
- AcMSSQLConnection class 99
- AcMultipleInputFilter class 33, 45
- AcOdaConnection class 200, 215
- AcOdaInterfaces.jar 205
- AcOdaSource class 200, 215
- AcODBCConnection class 99
- AcOracleConnection class 99
- AcSingleInputFilter class 33
- AcSQLQuerySource class 156
- AcSqlQuerySource class 154
- AcTextQuerySource class 154
- Actuate Basic
 - accessing stored procedures with 186–195
 - creating locale-specific reports and 243
 - customizing data sources and 38
 - developing with 28
 - implementing custom drivers with 200
 - mapping variable types for 187
 - throwing errors and 356
- Actuate Data Integration Service Connection
 - option 73, 74
- Actuate Data Integration Service Data Source
 - option 73, 75
- Actuate Foundation Class Library 28
- Actuate Foundation Classes 99
 - See also* classes
- ad hoc data access 72
- ad hoc data filters
 - See also* information objects
 - applying 244
- ad hoc parameters
 - See also* ad hoc data filters
 - adding to queries 150, 151
 - assigning values to 122
 - changing properties for 150
 - creating 149, 151

- ad hoc parameters (*continued*)
 - filtering data with 150
 - naming 151
 - restrictions for 181
 - setting properties for 151–152
 - viewing properties for 151
- add method 320
- adding
 - aggregate functions 116, 118
 - columns
 - to data rows 64
 - to designs 60
 - to queries 113, 116, 129
 - components 6, 96
 - conditions
 - to filters. *See* filter conditions
 - to queries 120–126, 147, 150, 151
 - configuration files to designs 17
 - connections 5, 8, 24, 95
 - controls to frames 175
 - custom data source builders 200
 - custom data sources 38, 80
 - data filters 46, 48, 50
 - data row variables 37, 61
 - data sources 24
 - default connections 19, 22
 - formulas 115, 116, 118
 - libraries to configurations 200
 - ODA drivers 200
 - parameters to expressions 146
 - sorting criteria 39, 40, 362
 - stored procedures 169–171, 175
 - tables to queries 106, 109
 - titles to reports 144
- addSortKey method 362
- AddToDriver property 222
- Adhoc Conditions page (Textual Query Editor) 151, 152
- Adhoc Parameters page (Textual Query Editor) 151
- Advanced Properties icon 68
- AFC. *See* Actuate Foundation Classes
- aggregate column names 125
- aggregate conditions 120
- aggregate controls 36
- aggregate expressions 117, 118
- aggregate functions
 - adding to queries 118
 - creating computed fields and 116
 - summarizing data and 118
- aggregate rows
 - adding to queries 116
 - creating 117–120
 - removing columns from 119
 - removing conditions on 125, 126
 - selecting columns for 117, 119
 - setting conditions for 120, 122, 124–126
- aggregation 32
- AIS connections 73, 74
 - See also* Integration service
- AIS data sources 73, 75
- Alias attribute (configurations) 19
- Alias Name dialog box 107
- aliases
 - table names and 107, 109
- alignment 246
- alignment values 246
- ALL constant 338
- ALL_LEVEL constant 338
- AllocateCursor method 99, 187
- alternate character sets 181
- alternate names 104
 - See also* aliases; display names
- alternative data types 220, 226
- AlternativeOdaDataTypes element 226
- AlternativeOdaDataTypes type 218
- application logger 343
- application servers
 - See also* servers
- applications
 - installing custom drivers and 236
 - logging messages for 343
 - ODA data sources and 202
 - providing column information for 239
- arguments. *See* parameters
- Arguments element 223
- ArrayOfInputFieldDefinition type 236
- ArrayOfInputParameterDefinition type 236
- ArrayOfNameValuePair type 237
- ArrayOfOutputFieldDefinition type 237
- ArrayOfOutputParameterDefinition type 238
- ArrayOfProperty type 238

- ArrayOfResultColumn type 238
- ArrayOfString type 218, 239
- arrays
 - input field definitions and 236
 - input parameter definitions and 236
 - name-value property pairs and 238
 - output field definitions and 237
 - output parameter definitions and 238
 - result set columns and 238
 - string elements and 218, 237, 239
- ascending sort order 361
- ASCII text files. *See* text files
- asterisk (*) character
 - parameters and 134
 - query statements and 130
- attributes. *See* properties
- Auto Contents group (Properties page) 68, 78
- Auto Joins command 107
- Automatic element 246
- AutoSort value 39
- Axis element 258
- AxisType element 240
- AxisType type 239

B

- backing up report files 159
- balloon help 258
- BAPI data sources
 - developing queries for 275
- BAPIs (business object APIs)
 - See also* BAPI data sources
- base64 encoding 262
- BigDecimal values 321, 332
- binary large object values. *See* BLOB values
- binary streams 273
- BindColumn method 187
- BindParameter method 157
- blank values 134
- BLOB data type 273
- Blob element
 - OdaDataType type 252
 - OdaScalarDataType type 253
- BLOB values 273, 274, 277, 309
- buffered output 367
- Builder icon 116
- BuildFromRow method 36
- building reports. *See* creating reports; designing reports
- By Filter setting 141
- By Query setting 141

C

- calculated columns or fields. *See* computed fields
- calculations
 - aggregate rows and 117, 118
 - computed fields and 36, 115
- CanSeek method 32
- CanSortDynamically method 38, 39
- case conversions 103
- case sensitivity
 - custom driver names 207
 - parameter names 149
 - string comparisons 103
- Center element 246
- Change Column Used dialog box 165
- changes
 - committing 290
 - rolling back 291
 - undoing 129
- changing
 - column names 136, 165
 - connections 23
 - data row variables 58, 60
 - data sources 24
 - default data processing 28
 - default display lengths 103, 138
 - display names 136
 - file paths 160
 - fonts 67, 68, 77, 78
 - formulas 116
 - graphical queries 113, 155, 156
 - ODA connection properties 204
 - ODA driver configurations 201, 210
 - result sets 135–138
 - stored procedure names 173
 - table names 164
 - textual queries 129, 131, 155
- character large object values. *See* CLOB values
- character sets 181
- character strings. *See* strings

- characters
 - CSV text files and 80
 - display names and unsupported 61
 - limits in string expressions 116, 118
 - truncated 61
- charts 32, 36
- check boxes 250
- Check Stored Procedure setting 180
- Choose Database page 19, 24
- class files 205
- class libraries 28
- class loader conflicts 224
- Class Variable dialog box 48, 81, 83
- classes
 - connection components and 99
 - custom drivers and 264
 - data row components and 34
 - design components and 28
 - development environment for 28
 - ODA drivers and 200, 201, 204, 205
 - ODA Java reference for 265
 - reusing 200
- ClassName property 222
- CLASSPATHs 223
- clear method 320
- Clear Query icon 130
- clearInParameter method 328
- client software 8, 94
- Clipboard 128
- CLOB data type 287
- Clob element
 - OdaDataType type 252
 - OdaScalarDataType type 253
- CLOB values 278, 287, 310
- close method
 - FileHandler 267
 - Handler 270
 - IConnection 289
 - IResultSet 308
 - IStatement 328
 - StreamHandler 267
- CLOSE_FAILURE constant 348
- closing
 - connections 99, 289
 - data adapters 32
 - files 32
 - input adapters 34
 - output streams 367
 - statements 328
 - text files 85
- code
 - See also* Actuate Basic
 - accessing text files and 5, 85
 - creating locale-specific reports and 243
 - customizing data streams and 31
 - executing stored procedures and 190
 - filtering data and 43, 46
 - implementing custom drivers with 200
 - referencing column names in 103, 138
 - sorting data and 40
 - throwing errors and 356
 - verifying input values with 150
- collections
 - adding rows to 319, 320
 - getting data source objects for 300
 - getting size 325
 - removing elements in 320
 - testing for elements in 320
- colon (:) character 146
 - query statements and 147, 151
- Column Editor
 - accessing 62, 65
 - changing columns with 62
 - changing queries and 137
 - overview 60
 - setting column attributes in 61, 65
- column headers 175
- column headings
 - data rows and 61
 - result sets and 258
- Column Name property 151
- column names
 - associating with parameters 151, 152
 - changing for query results 136, 165
 - defaults for 58
 - defining joins with 110
 - entering in SQL statements 150, 152
 - getting 61, 138, 316, 317, 363
 - locating table-specific 165
 - prompting for input and 150
 - referencing in code 103
 - setting fonts for 68, 77
 - truncated 54
- "Column not found" message 165

- Column Property page (Query Editor) 116
- Column Qualifiers command 105
- column widths 61, 107
- columnar report layouts 240
- ColumnAxisInfo type 239
- columnNoNulls constant 315
- columnNullable constant 315
- columnNullableUnknown constant 315
- columns
 - See also* fields; output columns
 - adding
 - to data row components 64
 - to queries 113, 116, 129
 - to reports 60
 - applying conditions to 120
 - binding to data row variables 34, 187
 - changing data types for 135, 138
 - changing properties for 62, 138
 - creating computed. *See* computed fields
 - defaults for 54
 - deleting from data rows 66
 - deleting from queries 114, 119
 - determining axis type 239
 - displaying data in 107, 114
 - entering formulas in 115, 116, 118
 - filtering on 244
 - freezing 108
 - getting date values for 310
 - getting decimal precision for 317
 - getting decimal values in 309
 - getting display lengths for 316
 - getting index of 308
 - getting number of 316
 - getting numeric values for 311
 - getting sort key 363
 - getting string values for 312
 - getting time values in 313
 - getting type 317
 - grouping on 119
 - joining tables and 109, 111
 - naming. *See* column headings; column names
 - referencing 133, 138, 165
 - renaming stored procedures and 174
 - reordering 58, 116
 - resizing 61, 103, 107, 138
 - returning from
 - queries 102, 130
 - result sets 238, 239, 256, 259
 - stored procedures 173, 180, 191
 - text files 83, 84
 - selecting multiple 114
 - setting dates for 321
 - setting decimal values for 321
 - setting display lengths for 58, 258
 - setting numeric values for 322, 323
 - setting string values for 323
 - setting time values for 324, 325
 - setting unique values for 111
 - setting values at run time 145
 - sorting data in 281, 361
 - sorting on 42, 118, 140, 141
 - summarizing data and 117
 - synchronizing queries and 102
 - testing for null values in 315, 318
 - trimming trailing spaces in 103
 - updating query 163
- Columns page (Query Editor) 113, 115, 116, 117, 118
- Columns page (Textual Query Editor) 138
- comma (,) character 80
- Command element 203, 241
- command line arguments (ODA drivers) 202, 223
- comma-separated values files. *See* CSV files
- commit method 289
- Compare method 42
- CompareKeys method 43
- comparisons 43, 103
- component libraries 24
- component palettes. *See* Toolbox
- Component Properties dialog box 9
- components
 - See also* specific type
 - accessing data and 5
 - adding 6, 96
 - developing 28
 - instantiating 30
 - publishing 7
 - referencing 24, 30
 - saving 55
 - subclassing 24
- computed columns. *See* computed fields
- computed data row variables 37, 60, 61, 62

- Computed Field command 115
- computed fields
 - building expressions for 62, 116
 - changing attributes of 60
 - creating 36, 113, 115–116
 - deleting 120
 - naming 115
 - setting conditions for 125
- computed values. *See* calculations
- concurrent connections 7
- conditional expressions 151
 - See also* Boolean values
- conditional sections 7, 55
- conditions
 - See also* filter conditions
 - adding to queries 120–126
 - aggregating data and 120, 124, 125, 126
 - applying at run time 124
 - deleting from queries 123, 124, 125, 126
 - placing on row retrieval 120, 123, 124
 - running stored procedures and 181
- Conditions page (Query Editor)
 - defining ad hoc parameters on 150
 - defining conditions on 122, 123
 - deleting conditions from 123, 124
- CONFIG constant 339
- CONFIG level messages 344
- config method 344
- CONFIG_LEVEL constant 339
- ConfigKey property 23
- Configuration file for connections
 - parameter 22
- configuration files
 - See also* configurations
 - accessing ODA 206
 - accessing sample 15
 - adding data sources to 210
 - changing 201, 210
 - creating 15, 207
 - defining schemas for 210
 - elements in 16, 19, 22, 23
 - including in designs 17
 - installing 201, 210
 - loading 201, 207
 - overview 12
 - removing from designs 14
 - selecting 13
 - setting connections and 15, 20, 22
 - setting run-time connections and 22
 - specifying data sources in 24
 - specifying paths for 18
 - updating 201
- configurations
 - flat file data sources and 215
 - localized databases and 91
 - ODA data sources and 201, 202, 206
 - ODA drivers and 201, 227
 - ODA log files and 231, 293
 - ODBC data sources and 91
 - predefined data sources and 24, 210
- connection classes 99, 201
- connection components
 - See also* connections
 - adding to configuration files 24
 - adding to designs 7, 8, 19, 22, 95
 - changing functionality of 28
 - defined 5
 - instantiating 28
 - omitting 5, 7
 - placing in data streams 6, 28
 - placing in sections 5, 7, 28
 - redefining methods for 28
- Connection element 19, 228
- connection factories 292
- connection information 7, 94, 99
- connection interface 289
- Connection slot 5, 8, 26, 95
- connection strings 91
- Connection type 218
- ConnectionProperties type 240
- connections
 - See also* connection components
 - accessing
 - custom data sources and 38
 - databases and 7, 94, 95
 - information objects and 73
 - ODA data sources and 203, 289
 - ODBC data sources and 90, 94, 95
 - text file data sources and 5, 81
 - changing 23, 204
 - closing 99, 289
 - creating 7–9, 12, 28, 74, 90
 - customizing 28
 - defining multiple 7, 23

- deploying executable files and 7
- determining if opened 291
- displaying 9, 170
- failing 99
- generating queries and 104, 130, 159
- generating reports and 4
- getting definitions for 243
- getting instances of 292, 295, 299
- getting metadata for 290
- getting number of active 296
- getting number of statements for 296
- inheriting 7
- installing database clients for 94
- installing ODA drivers and 200, 218, 294
- opening 28, 291
- overriding 12
- overview 6–7
- precedence for 7, 22
- properties files and 12
- selecting 24, 28, 96
- setting properties for 9, 94, 96, 240
- sharing 7
- specifying default 20, 22
- specifying run-time 22
- verifying 159, 160
- ConnectOptions element (configurations) 23
- Content frames 175
- Content slot 175
- context menus 129
- ContinueBuilding value 36
- control type constants (input) 248, 250
- ControlCheckBox value 250
- ControlList value 248, 250
- ControlListAllowNew value 248, 250
- ControlRadioButton value 250
- controls
 - See also* specific type
 - adding to frames 175
 - changing data row types and 61
 - prompting for input and 248, 250
 - setting values for 36
- ControlTextBox value 248, 250
- ControlType element 250
- conversions 220
- converting graphical queries to textual 131–133, 156
- copying
 - query statements 128
 - report files 159
- copyright information 233
- CopyrightNotice element 233
- CPointer data type 190, 192
- Create New Data Source dialog box 92
- createLogger method 206, 351
- createStatement method 290
- creating
 - aggregate expressions 117, 118
 - aggregate rows 117–120
 - application logger 343
 - computed data row variables 37
 - computed fields 36, 113, 115–116
 - conditional expressions 125, 126
 - configuration files 15, 207
 - connections 7–9, 28, 90
 - custom data source builder 200
 - custom data sources 38, 43, 80
 - custom drivers 200, 264
 - data adapters 28, 30, 33
 - data filters 46
 - data rows 34
 - database cursors 187
 - executable files 54
 - graphical queries 98, 102, 103
 - input filters 46
 - input parameters 186, 248
 - joins 109, 110
 - multi-table reports 46
 - ODA drivers 200
 - ODBC data sources 90
 - Oracle stored functions 181
 - Oracle stored procedures 180, 181
 - output parameters 187, 255
 - predefined connections 12
 - predefined data sources 24, 210
 - query data sources 96–98
 - report parameters 144, 149
 - report titles 144
 - sort filters 42, 141
 - SQL queries. *See* queries
 - statement objects 290
 - summary data 117, 118
 - textual queries 98, 102, 128–131
 - union filters 50

- cross tabulation. *See* crosstabs
- crosstabs
 - defining default layout for 240
- CSV files
 - accessing data in 80
 - closing 85
 - customizing data sources for 80
 - defining data row variables for 83
 - displaying data from 84, 86
 - opening 85
 - retrieving data from 85
- cubes. *See* data cubes
- Currency data type 187
- cursor parameters 181
- cursor variables
 - fetching rows from 193
 - output parameters as 190
 - restrictions for 192
 - returning from stored functions 181
 - types described 190
- cursors
 - closing 308
 - creating 187
 - defined 99
 - executing queries and 157
 - moving 314, 320, 321, 330
 - opening 32
 - setting position of 307
- custom data drivers 200
- custom data sources 38, 43, 80
 - See also* ODA data sources
- Custom From Clause command 113
- Custom FROM Clause dialog box 113
- customizing
 - connections 28
 - data drivers 200, 236, 264
 - data filters 32, 34, 45
 - data rows 34, 35
 - data sources 38, 43, 80, 219
 - data streams 7, 31–33
 - default data processing 28
 - ODA data source builder 200
 - queries 112, 157
 - reports 28
 - stored procedures 186–195

D

- data
 - See also* values
 - accessing 4, 28, 32, 90
 - aligning 246
 - changing default processing for 28, 32
 - combining from multiple rows 117
 - customizing drivers for 200, 264
 - defining default alignment for 246
 - displaying 54, 107, 114
 - filtering. *See* filtering data; filters
 - formatting. *See* formats; formatting
 - presorting 40, 41, 141
 - previewing 107, 114, 154
 - removing conditions on 123, 124
 - retrieving from
 - custom data streams 31, 32
 - databases 90
 - external data sources 33
 - information objects 73
 - multiple data sources 45
 - ODA data sources 203, 204
 - ODBC data sources 90, 144
 - predefined data sources 9
 - stored procedures 90, 128, 175, 187, 188
 - text files 80, 85
 - retrieving with conditions 120, 122, 123, 147
 - returning specific values for 145, 146
 - selecting 9, 32, 120
 - sorting. *See* sorting data
 - summarizing 116, 117–120
- data adapter components 6
- data adapters
 - accessing multiple 45
 - accessing non-sequential streams and 32
 - changing input records from 32
 - closing 32
 - creating data rows and 34
 - creating input filters for 33, 45
 - instantiating 28, 30, 33
 - opening 32, 33
 - sorting data and 42
- Data command 104
- data components 6
 - See also* components; data

- data controls 35
 - See also* controls; data
- data dictionary 110, 158, 161
- data filter classes 33
- data filter components
 - adding to data streams 29, 33
 - adding to designs 44, 46
 - as transient objects 29
 - changing methods for 31
 - defined 5
- data filters
 - defining parameters as 144
 - getting 270
 - setting at run time 144
 - setting log handler 272
- Data Integration Option 45
- Data Integration Service Connection
 - option 73, 74
- Data Integration Service Data Source
 - option 73, 75
- data object executable files 54, 55
 - See also* information objects
- data row components
 - See also* rows
 - adding to designs 96, 97
 - as transient objects 29
 - customizing 35
 - defined 5
 - displaying 58
 - placing variables in 61
- Data Row Editor
 - accessing 59
 - changing column properties from 62
 - changing display names and 136
 - displaying data rows with 58
 - removing columns with 66
 - reordering columns from 59
 - selecting columns with 64
- Data Row Editor command 59
- data row objects 34, 187
 - See also* rows
- data row variables
 - adding to designs 61
 - binding columns to 34, 187
 - changing 60
 - creating computed fields with 37, 60, 113
 - defined 34
 - defining for
 - custom data sources 32, 38
 - text file data sources 83
 - displaying 59, 138
 - getting column names for 316
 - setting column attributes for 61
 - setting data types for 61
 - viewing attributes of 60
- data rows. *See* rows
- data source builder. *See* ODA data source builder
- data source components
 - See also* data sources
 - adding to configuration files 24
 - adding to report sections 99
 - as transient objects 29
 - changing methods for 31
 - defined 6
 - publishing 7
 - subclassing 24
- data source names 91, 94, 300
- data source variables 58
- data sources
 - See also* data source components; databases; specific type
 - accessing 38, 50, 200
 - adding to configuration files 210
 - adding to table names 105
 - changing 24
 - combining data from multiple 45, 46, 48
 - committing changes to 290
 - configuring connections for 19, 20, 22
 - connecting to. *See* connections
 - creating custom drivers for 200, 236, 264
 - creating data streams for 7, 29
 - creating query 96–98
 - creating sort filters for 39, 42, 139
 - customizing 38, 43, 80, 219
 - defining information objects as 73, 75
 - defining stored procedures as 169
 - determining parameter type for 301, 302
 - displaying data rows in 54, 55, 58
 - displaying system 92
 - getting collections of 300
 - getting information about 297
 - getting type 305
 - getting version of 299, 300

- data sources (*continued*)
 - handling errors for 356
 - installing client software for 8
 - limiting data returned by 43
 - maintaining query information for 154, 156
 - mapping data types to 135
 - predefining 24, 210
 - referencing 159
 - retrieving data from 4, 6, 9, 32, 45
 - retrieving input records from 34
 - returning sort mode for 301
 - selecting 200, 210
 - setting properties for 20, 23
 - sorting data and 39, 40
 - synchronizing queries and 102, 156, 161, 163
 - updating 156
 - verifying connections for 159, 160
- data stream components 6, 97, 201, 202, 236
- data stream definition files 203
- data stream definitions 203, 240, 243
- data stream slots 95
- data stream types 242
- data streams
 - accessing
 - stored procedures and 169
 - adding connection components to 6, 28
 - adding multiple data adapters to 30
 - building 29
 - creating custom data sources for 38, 80, 203
 - creating SQL queries and 96
 - customizing 7, 31–33
 - filtering data in 43, 44, 46
 - retrieving data and 5, 32
 - setting properties for 203, 240
 - sorting data in 40
- data structures 319
- Data Toolbox 6
- data types
 - accessing multiple data sources and 48, 51
 - accessing stored functions and 181
 - aggregating data and 116, 118
 - assigning to
 - data row variables 61
 - ODA data sources 207, 220, 221, 226, 252, 253
 - parameters 145, 148
 - stored procedures 178, 187
 - text files 83
 - changing column 103, 135, 138
 - changing data row variable 61
 - changing static parameter 149
 - creating SQL queries and 135
 - getting native data source 138
 - mapping external 207, 220
 - mapping to
 - native data source types 61
 - Oracle databases 180
 - maximum precision allowed for 305
 - synchronizing queries and 159
- Database Browser 104, 105, 106
- database clients 8, 94, 159
- Database Connection command 24, 94
- database connection components 5, 7, 8
 - See also* connection components
- Database Connection dialog box 24
- database connection types 94
- database cursors
 - closing 308
 - creating 187
 - defined 99
 - executing queries and 157
 - moving 314, 320, 321, 330
 - opening 32
 - setting position of 307
- Database Login dialog box 9, 130
- database schemas
 - accessing data and 90
 - determining if current 162
 - updating 158, 161
- database servers. *See* servers
- databases
 - See also* data sources
 - accessing data in 32, 90, 99, 102
 - accessing with stored procedures 168
 - calling stored procedures in 168, 169, 186
 - choosing 24, 93
 - connecting to 7, 24, 94, 95, 99
 - creating queries for 102, 154
 - displaying stored procedures in 171, 174
 - filtering data from 33, 144

- installing localized 91
- instantiating data row objects for 34
- joining multiple tables in 110
- limiting rows returned from 120
- logging into 95
- removing obsolete tables in 164
- restricting data retrieval from 144
- retrieving data from 90
- running consistency checks for 160
- selecting tables from 104, 106
- setting paths for 160
- sorting data from 39, 139
- synchronizing stored procedures in 168, 180
- updating 163
- verifying connections for 159, 160
- DataFont property 68, 77
- DataRow components. *See* data row components
- DataSource components. *See* data source components
- DataSource element 220
- DataSource type 219
- DataSources element 228
- DataSources type 220
- DataStream components. *See* data stream components
- DataStreamDefinition element
 - DesignSessionRequest type 203, 204, 242
 - DesignSessionResponse type 203, 204, 244
- DataStreamDefinition type 240
- DataType element
 - InputFieldDefinition type 247
 - InputParameterDefinition type 250
- DataTypeMapping element 220, 221, 252
- DataTypeMappings type 221
- DataValue variable 35
- Date data type 187, 321, 332
- Date element
 - OdaDataType type 252
 - OdaScalarDataType type 254
- dates
 - adding to page footers 68, 77
 - computing values for 37
 - getting 278, 310
 - setting 321, 332
- DB2 databases
 - See also* databases
 - accessing stored procedures in 168
 - connecting to 8, 94, 99
 - creating queries for 103
 - retrieving data from 9
- debugging ODA drivers 202
- Decimal element
 - OdaDataType type 253
 - OdaScalarDataType type 254
- decimal values
 - getting column-specific 309
 - getting from output parameters 277
 - getting number of digits in 306, 318
 - getting precision of 305, 317
 - setting 321, 332
- default alignment 246
- default configurations 13
- default connections 19, 22
- default data types 149
- default page layouts 54
- default sort order 103
- default values
 - ODA data sources and 247, 250
 - report parameters and 134
 - static parameters and 146
- DefaultValue element
 - InputFieldDefinition type 247
 - InputParameterDefinition type 250
- DefinedIn attribute (configurations) 19
- DefineProcedureInputParameter method 186
- DefineProcedureOutputParameter method 187
- DELETE statements 157
- deleting
 - columns 66, 114, 119
 - computed fields 120
 - joins 109, 112
 - tables from queries 109
- delimited text strings 369, 372, 373
- deploying executable files 7
- descending sort order 43, 140, 361
- Describe Query icon 130
- Describe Query operations 130, 137, 148, 151
- Description attribute (configurations) 19
- Description element
 - InputFieldDefinition type 247

- Description element (*continued*)
 - InputParameterDefinition type 250
 - OutputFieldDefinition type 254
 - ResultColumn type 258
- design components 28
- design files. *See* report object design files
- design tools 200, 236
 - See also* specific Actuate designer; ODA design tool
- DesignerName element 222
- DesignerName property 220
- DesignerSpecific element 221
- DesignerSpecific type 221
- DesignerSpecificProperties element 219, 220
- DesignerSpecificProperties type 221
- DesignerSpecificType element 222
- designing reports 4, 14
 - See also* designs
- DesignLibrary property 222
- designs
 - See also* page layouts
 - accessing
 - ODA data sources and 201, 204, 240
 - accessing stored procedures for 169–171, 175
 - adding
 - connections to 19, 22
 - building computed values for 36
 - building custom data sources and 202
 - creating xv
 - creating connections in 7–9, 95
 - creating SQL queries and 97
 - defining alternative data types for 220, 226
 - developing 28
 - including configurations in 17
 - installing custom drivers for 200, 206
 - migrating to production environments 23
 - referencing components in 30
 - referencing data sources in 159
 - removing default configurations for 14
 - retrieving data for 9
 - sample configurations for 15
 - saving 38
 - selecting data sources for 24, 84, 200, 210
 - viewing data row variables in 58, 59
- DesignSessionRequest element 203
- DesignSessionRequest type 242
- DesignSessionResponse element 203, 204
- DesignSessionResponse type 243
- design-time interface 204, 222, 228
- DesignTimeInterface element 202, 228
- DesignTimeInterface type 222
- developing reports 13, 39
- development environment 28
- DimensionAttribute element 239
- DimensionMember element 239
- directory paths
 - component libraries and 19
 - configuration files 14
 - connection configurations and 18
 - custom drivers and 207
 - data source drivers 160
 - executable files 223
 - file-based data sources 160
 - log files 232
 - ODA design tools 222
 - ODA drivers 208, 225
- disconnecting from databases 99
- Disk-based Data Row Sorter setting 42
- display formats. *See* formats; formatting
- Display length field 138
- Display Name property 210
- display names
 - See also* DisplayName element
 - column headings 61
 - columns 136
 - data source lists 210, 220
 - ODA connections 219
 - ODA input parameters 248, 250
 - ODA output parameters 254
 - property values 230
 - result set column names 245, 258
- DisplayFormat element 258
- displaying
 - column names 165
 - connection properties 9, 169
 - data 54, 107, 114
 - data row variables 59, 138
 - data rows 54, 55, 56, 58
 - default values 134
 - nested sections 55
 - ODBC drivers 92
 - owner information 105
 - performance statistics 134

- query statements 127
- sample configurations 15
- SQL statements 126, 154
- stored functions 181
- stored procedures 171, 174
- system data sources 92
- unformatted data 55
- DisplayLength element 258
- DisplayName element
 - DataSource type 220
 - InputFieldDefinition type 248
 - InputParameterDefinition type 250
 - ODA connections and 219
 - OutputFieldDefinition type 254
 - PropertyType type 230
 - ResultColumn type 258
- DisplayNameColumn element 245
- distinct values 114
- documents. *See* reports
- Double data type 187, 322, 333
- Double element
 - OdaDataType type 253
 - OdaScalarDataType type 254
- double quotation mark (") character
 - CSV text files and 80
- double values
 - getting 279, 311
 - setting 322, 333
- .dox files. *See* data object executable files
- driver libraries 223
- DriverInitEntryPoint element 206, 231
- DriverLibraries element 231
- DriverLibraries type 223
- DriverName element 207, 228
- drivers
 - accessing configuration files for 206
 - accessing data with custom 200, 236
 - accessing ODBC data sources and 90
 - accessing sample ODA 200
 - backward compatibility for 228
 - calling statements for 275
 - changing configurations for 201, 210
 - connecting to ODBC data sources and 91, 93
 - creating joins and 110
 - customizing 200, 236, 264
 - customizing properties for 222
 - debugging custom 202
 - defining operating system for 228
 - developing ODA 224
 - displaying installed 92
 - getting metadata for 290
 - getting names of 295
 - getting number of connections for 296
 - getting number of statements for 296
 - getting version 295
 - installing ODA 200, 201, 207, 294
 - loading custom 207
 - naming custom 207, 228
 - running ODA 201
 - running predefined 200
 - setting exceptions for 356
 - setting logging configurations for 231, 293
 - unsupported features and 297
 - unsupported statements and 326
- DROP TABLE statements 157
- DSNs (data source names) 91
- dynamic sorting 38, 39
- DynamicCondition element 250
- DynamicConditionReference type 244
- DynamicQuery type 245
- DynamicValueList element 250

E

- e.Report Designer Professional
 - conditionalized queries and 120, 122
 - custom drivers and 200
 - data-processing functionality for 28
 - nonstandard installations and 160
 - ODA data sources and 201
 - sample configuration file for 13
 - supported connections for 8
 - supported data sources for 9
 - supported databases for 168
 - synchronization tools in 158
 - text file data sources and 5, 85
 - undefined parameters and 145
 - verifying input and 150
- e.reporting servers. *See* iServer; servers
- e.Spreadsheet reports. *See* spreadsheet reports
- e.Spreadsheet server. *See* iServer
- Edit SQL tab 132

- Editable value 260
- editors 15
 - See also* specific Actuate editor
- elements (XML) 218, 236
- Enable Group By and Having editors
 - setting 119
- encapsulated SQL queries 72
- encoding 262
- Encyclopedia volumes 76
 - accessing information objects in 72
- equals method 340
- equi-joins 110
- error codes 349, 356
- error constants 348
- error messages
 - See also* errors
 - connecting to data sources and 99
 - creating 356
 - logging 346, 347
 - outputting 349
- error method 349
- errors
 - See also* exceptions
 - failed connections and 99
 - log records and 354, 355
 - LoggingErrorHandler and 271, 272, 348
 - ODA data sources and 356
 - SQL queries and 119
 - unsupported methods and 264
- example ODA driver 200
- exception class 356
- exceptions
 - See also* errors
 - getting 354
 - initializing 359
 - logging 346, 347, 348
 - outputting 349
 - providing information with 356
 - returning cause of 358
 - throwing 355
- Executable element 223
- executable files
 - building ODA data sources and 202, 222, 228
 - creating 54
 - deploying 7
 - generating queries and 114
 - setting paths for 223
- Execute method 157, 187
- execute method 328
- execute privilege 73
- executeQuery method 329
- executing queries 157, 329, 334, 335
- executing reports 201
- executing stored procedures 187
- experts. *See* wizards
- export parameters (RFMs) 32
- expression builder 116, 123
- expressions
 - adding parameters to 146, 151
 - aggregating data and 124, 125
 - character limits for 116
 - creating computed fields and 36, 115, 116
 - creating queries and 120, 122
 - entering invalid 150
 - entering run-time 123, 149
 - filtering data and 150
 - leading spaces in 151
 - retrieving data rows and 62
 - summarizing data with 117, 118
- extensible markup language. *See* XML
- external data sources 33
- external data types 206
- external exceptions 356
- ExternalDesignState element
 - DesignSessionRequest type 204, 242
 - DesignSessionResponse type 204, 244
- ExternalState type 245

F

- Factory service
 - ODA data sources and 205, 206, 207
 - ODA drivers and 201
 - transient objects and 29
- Fetch method
 - accessing multiple data sources and 45, 48, 51
 - accessing stored procedures and 187, 193
 - building custom data streams and 32
 - displaying CSV data and 84, 85
 - filtering data rows and 33, 34, 44, 45
 - instantiating data rows and 34
- fetch position 33

- fetching data. *See* Fetch method
- FetchLimit property 131
- field definitions
 - creating 246, 254
 - listing input parameter 236
 - listing output parameter 237
- Field List 175
- field names. *See* column names; column headings
- FieldControlType element 248
- fields
 - See also* columns; computed fields
 - changing data row variables and 58
 - changing default values for 36
 - comparing values in 43
 - defining class variables for 83
 - displaying database 107
 - displaying variables for 84
 - limiting number returned 9
 - restricting retrieval of 144
 - retrieving with data row variables 34
 - selecting 18, 65
 - specifying as primary keys 111
- Fields command 84, 175
- Fields dialog box 84
- file DSNs 91
- file names 81, 82, 85
- file numbers 81
- file paths. *See* paths
- file-based databases 159, 160
- FileHandler class 205, 266
- files
 - See also* specific file type
 - accessing ODA data sources and 200
 - backing up 159
 - closing 32
 - configuring environments and. *See* configuration files
 - copying report 159
 - creating configuration 207
 - loading configuration 13
 - saving temporary 202
 - specifying connection properties in 12
- Filter class 205
- filter classes 33
- filter conditions
 - See also* filtering data; filters
 - adding to graphical queries 148, 150
 - adding to textual queries 147, 151
 - defining ad hoc 150, 151
 - defining static 145, 147, 148
- Filter interface 268
- filtering data
 - databases and 144
 - graphical queries and 148, 150
 - guidelines for 43
 - information objects and 144
 - input sources and 29, 33, 44
 - multiple data sources and 45
 - non-sequential data streams and 32
 - ODA data sources and 244
 - ODBC data sources and 144
 - SQL queries and 103, 132, 144
 - text file data sources and 86
 - textual queries and 147
- filters
 - See also* data filter components
 - creating 5, 43, 46, 50
 - customizing 32, 34, 45
 - defining conditions for. *See* filter conditions
 - defining parameters as 144, 147, 150
 - getting 270
 - instantiating data rows for 34
 - overview 33
 - reusing 43
 - setting at run time 144, 145, 149
 - setting log handler 272
 - sorting data with 39, 42, 139, 141
 - specifying input sources for 44
 - specifying parameters as 150
- finalize method 367
- findColumn method 308
- finding
 - embedded strings 369
 - stored procedures 174
- findInParameter method 329
- findOutParameter method 276
- FINE constant 339
- FINE level messages 344
- fine method 344
- FINE_LEVEL constant 339
- FINER constant 339
- FINER level messages 344

- finer method 344
- FINER_LEVEL constant 339
- FINEST constant 339
- FINEST level messages 344
- finest method 344
- FINEST_LEVEL constant 339
- Finish method
 - creating custom data streams and 32
 - displaying CSV data and 84, 85
 - filtering data and 34
 - generating content with 36
- FinishedBuilding value 36
- flat file data sources 40, 41, 139, 215
- FlatFileExample directory 200
- floating point numbers
 - getting 279, 311
 - setting values 322, 333
- flush method
 - Handler 270
 - StreamHandler 367
- FLUSH_FAILURE constant 348
- fonts
 - changing attributes of 67, 68
 - changing default 78
 - changing query output 77–78
 - setting properties for 77
- footers 68, 77
- foreign keys 110
 - See also* joins
- format method
 - LogFormatter 341
 - SimpleFormatter 360
- FORMAT_FAILURE constant 348
- formats 55, 61, 258
- FormattedReport value 55
- formatting
 - log records 341, 360
 - query output 77–78
 - strings 341, 360
 - table names 109
 - view names 109
- formulas 115, 116, 118
 - See also* expressions
- frames 175
- freezing columns and rows 108
- FROM clause 106, 112
 - See also* SQL statements

- functions
 - See also* methods
 - stored procedures and 173
 - summary data and 116
- fundamental data types. *See* data types

G

- General command (Options) 13
- General page (Options) 13
- generating
 - error messages 99
 - executable files 55
 - reports 4
- GENERIC_FAILURE constant 348
- getBigDecimal method
 - ICallStatement 277
 - IResultSet 308
- getBinaryStream method 273
- getBlob method
 - ICallStatement 277
 - IResultSet 278, 309, 310
- getBytes method 273
- getCause method 358
- getCharacterStream method 287
- getColumnCount method 316
- getColumnDisplayLength method 316
- getColumnLabel method 316
- columnName method 317
- getColumnType method 317
- getColumnTypeName method 317
- getConnection method
 - IConnectionFactory 292
 - IConnectionMetaData 295
 - IDataSourceMetaData 299
- getDataSourceMajorVersion method 299
- getDataSourceMinorVersion method 299
- getDataSourceObjects method 299
- getDataSourceProductName method 300
- getDataSourceProductVersion method 300
- getDate method
 - ICallStatement 278
 - IResultSet 310
- getDelimitedStringCount method 369
- getDouble method
 - ICallStatement 279
 - IResultSet 311

- getDriverMajorVersion method 295
- getDriverMinorVersion method 295
- getDriverName method 295
- getDriverVersion method 295
- getErrorCode method 358
- getFilter method 270
- getFormatter method 270
- getHandler method 345
- getInt method
 - ICallStatement 279
 - IResultSet 311
- getLevel method
 - Handler 270
 - Logger 345
 - LogRecord 353
- getLogger method 351
- getLoggingErrorHandler method 271
- getMaxConnections method 296
- getMaxRows method 329
- getMaxStatements method 296
- getMessage method 354
- getMetaData method
 - IConnection 290
 - IResultSet 312
 - IStatement 330
- getMetaDataOf method 280
- getMillis method 354
- getMoreResults method 330
- getName method
 - Level 340
 - Logger 345
- getNextException method 359
- getOdaMajorVersion method 296
- getOdaMinorVersion method 296
- GetOutputParameter method 187, 192
- getParameterCount method 304
- getParameterMetaData method 330
- getParameterMode method 304
- getParameterType method
 - IParameterMetaData 305
 - IStatement 331
- getParameterTypeName method 305
- GetPosition method 32
- getPrecision method
 - IParameterMetaData 305
 - IResultSetMetaData 317
- GetProcedureStatus method 187, 192
- getResultSet method
 - ICallStatement 280
 - IStatement 331
- getResultSetNames method 281
- getRow method
 - ICallStatement 281
 - IResultSet 312
- getScale method
 - IParameterMetaData 306
 - IResultSetMetaData 318
- getSortColumn method 363
- getSortColumns method 363
- getSortKeyCount method 364
- getSortMode method
 - IDataSourceMetaData 300
 - SortSpec 364
- getSortOrder method 364
- getSortSpec method
 - ICallStatement 281
 - IStatement 331
- getSQLState method 359
- getSQLStateType method 301
- getString method
 - ICallStatement 282
 - IResultSet 312
- getSubString method 287
- getThrown method 354
- getTime method
 - ICallStatement 282
 - IResultSet 313
- getTimestamp method
 - ICallStatement 283
 - IResultSet 313
- global parameters 145
- global reporting solutions. *See* locales
- graphical queries
 - See also* Query Editor
 - changing result sets for 135
 - changing statements for 113, 155, 156
 - converting to text-based 97, 131–133, 156
 - creating 98, 102, 103
 - deleting conditions for 124, 126
 - displaying statements for 154
 - filtering data with 103, 150
 - previewing result sets for 133
 - removing columns from 114
 - reordering columns for 116
 - returning computed fields with 115

- graphical queries (*continued*)
 - selecting columns for 113
 - selecting tables for 104, 106
 - setting conditions for 120, 123, 126, 148, 150
 - sorting with 140, 142
 - summarizing data with 117, 118
 - synchronizing 156, 158, 159, 162, 163
 - tracking changes to 164
 - updating 158, 163
- graphical query data sources 97
 - See also* query data sources
- graphical user interfaces. *See* user interfaces
- graphs. *See* charts
- GROUP BY clause 117
 - See also* SQL statements
- Group By page (Query Editor) 117, 118, 119
- Group By tab 119
- Group element 250
- group sections
 - creating queries for 40, 141
 - displaying unformatted data and 55
 - placing connections in 7
 - sorting data in 40
 - sorting defaults and 39, 139
- grouping column (Query Editor) 117, 118, 119
- grouping conflicts 117
- grouping data rows 117
- grouping information, preserving 55
- GUIs. *See* user interfaces

H

- Handler class 205, 269
- hashCode method 340
- HAVING clause 124
 - See also* SQL statements
- Having page (Query Editor)
 - displaying 125
 - removing conditions from 125, 126
 - setting conditions on 120, 122, 125, 126
- Heading element 258
- HelpText element 258
- HorizontalAlignment element 258
- HorizontalAlignment type 246

I

- IBlob interface 273
- ICallStatement interface 205, 275
- IClob interface 287
- IConnection interface 205, 206, 289
- IConnectionFactory interface 292
- IConnectionMetaData interface 205, 294
- IConnectionMetaData method 264
- icons 28, 181
- IDataSourceMetaData interface 205, 297
- IDataSourceMetaData method 264
- Identifier element 244
- IdentifierNativeTypeCode element 244
- IDriver interface 264
- Include element (configurations) 17
- INFO constant 339
- INFO level messages 345
- info method 345
- INFO_LEVEL constant 339
- InfoObjects. *See* information objects
- Information Object Data Source Editor 76
- Information Object Designer
 - multiple data sources and 45
- information objects
 - accessing data in 73
 - connecting to 73, 74
 - defining as data sources 73, 75
 - defining result set columns for 239
 - filtering data for 144
 - overview 72, 73
 - retrieving data and 45
- InformationObject value 54
- inherited parameters 145
- initCause method 359
- inner joins 110, 111
- input
 - NCHAR data types and 181
 - prompting for 145, 149
- input adapters 33, 45, 48
- input filter classes 33
- input filters 33, 44, 45, 46, 50
- input parameters
 - See also* stored procedures
 - appending rows to 320
 - checking for 178
 - creating 186, 248

- defining dynamic queries for 245
- determining if supported 301
- entering sample values for 172, 178
- finding index of 329
- generating names for 176
- getting data source type 305
- getting data type of 305, 331
- getting number of 304
- getting numeric precision for 305
- listing definitions for 236, 246
- mapping to report parameters 203, 241
- overview 176
- prompting for 175
- returning field definitions for 236
- running stored procedures and 181
- setting date values for 332
- setting decimal values for 332
- setting default values for 250
- setting numeric values for 333
- setting single row for 284
- setting string values for 335
- setting time stamps for 337
- setting time values for 336
- specifying as type 176, 177
- specifying single row for 284
- specifying tables with 284
- testing for null values in 306
- input sources
 - See also* data sources
 - accessing data in 32
 - accessing multiple 29
 - filtering data in 29, 33, 43, 44
 - opening 32
 - retrieving data from 28, 34
 - sorting data in 46
- InputFieldDefinition element 236
- InputFieldDefinition type 246
- InputParameterDefinition element 237
- InputParameterDefinition type 248
- InputParameters element 203, 241
- installation
 - configuration files 201, 210
 - localized databases 91
 - nonstandard locations 160
 - ODA drivers 200, 201, 207–208
- Integer data type 187, 323, 333
- Integer element
 - OdaDataType type 253
 - OdaScalarDataType type 254
- integers
 - See also* numbers
 - getting 279, 311
 - setting values for 323, 333
- Integration service
 - connecting to 73
- interfaces
 - See also* user interfaces
 - custom drivers and 200, 264
 - ODA drivers and 205, 264
 - ODA Java reference for 265
 - predefined data sources and 200
- InterfaceType element 231
- InterfaceType type 224
- international reporting solutions. *See* locales
- intValue method 340
- invalid expressions 150
- invalid values 134, 150, 314
- IParameterMetaData interface 205, 303
- IParameterMetaData method 264
- IRResultSet interface 205, 307
- IRResultSetMetaData interface 205, 315
- IRowSet interface 205, 319
- IsDefault attribute (configurations) 20, 22
- IsDefault property 20
- isEmpty method 320
- IsEnabled element 245
- iServer
 - See also* servers
 - accessing information objects and 73
 - defining connections for 7
 - installing custom drivers for 207, 210
 - installing ODA drivers for 208
- IsHidden element
 - InputFieldDefinition type 248
 - InputParameterDefinition type 250
- isLoggable method
 - Filter 268
 - Handler 271
 - Logger 345
 - StreamHandler 367
- isNullable method
 - IParameterMetaData 306
 - IRResultSetMetaData 318

- isOpened method 291
- IsPassword element 250
- IsRequired element
 - InputFieldDefinition type 248
 - InputParameterDefinition type 250
- IStatement interface 205, 326

J

- Java classes 264
- Java interfaces 205, 264
- Java Virtual Machines. *See* JVMs
- JavaLogFormatterClass element 232
- join operator icon 111
- join operators 111
- Join Properties page (Query Editor) 111
- joins
 - accessing data and 90
 - adding primary keys for 110
 - creating 109, 110
 - deleting 112
 - disabling automatic generation of 107
 - emulating 46
 - specifying type 110, 111

K

- KeepOdaRequestResponseFiles registry key 202
- keys. *See* joins; sort keys

L

- label controls 36
- LabelFont property 68, 77
- labels 36, 77, 78
- layouts 54, 240
 - See also* designs
- leading spaces 151
- Left element 246
- left outer joins 110
- length method
 - IBlob 274
 - IClob 288
- Level class 206, 338
- libraries
 - adding to configuration files 17, 18, 200
 - defining connections in 20
 - defining specific operating system 225

- developing with 28
- implementing custom drivers in 200, 223
- loading ODA driver 231
- publishing to 43
- referencing components in 24
- reusing classes in 200
- running queries from 40, 41
- setting paths for 19
- setting paths to ODA design 222
- LibrariesForOS element 224, 231
- LibrariesForOs element 224
- LibrariesForOsType type 225
- library files. *See* report object library files
- LibraryName element 225
- line controls 36
- Lineage element 258
- linking related tables 109
- list controls 248, 250
- ListOfProperties element 222
- ListOfProperties type 225
- lists 122
- loading
 - configuration files 13, 201, 207
 - custom data drivers 207
 - ODA driver libraries 231
 - text files 85
- local connections 4
- local parameters 145, 177
- locales
 - developing user interfaces for 204, 260
 - installing localized databases for 91
 - specifying 243
- locating
 - embedded strings 369
 - stored procedures 174
- Location element 225
- log files
 - configuring 231, 293
 - creating error handler for 348
 - filtering 268, 270, 272
 - formatting records for 341, 360
 - getting logging level for 354
 - getting messages in 354
 - publishing records to 267, 268, 271
 - setting formatter for 272, 368
 - setting logging levels for 354
 - streaming records for 366

- writing application messages to 343
- log handler 269
- log method 346
- log records 341, 360
- LogDirectory element 232
- LogFilenamePrefix element 232
- LogFormatter class 206, 341
- LogFormatter objects 270
- logger 206, 343
- Logger class 206, 343
- Logger objects 350, 351, 353
- logging constants 338
- logging error handler 348
- logging information (ODA configurations) 206
- logging into
 - databases 95
- logging levels 270, 272, 338
- logging package 231, 293
- LoggingErrorHandler class 206, 348
- logical values
 - See also* Boolean values
- login failures 256
- LoginFailed element 256
- LogLevel element 232
- LogManager class 206, 350
- LogRecord class 206, 353
- Long data type 187
- LOWER function 103

M

- mapping
 - ODA data types 207, 220, 221, 226, 252, 253
- mapping variable types 187
- maps (information objects) 72
- matrix reports. *See* crosstabs
- MDX queries
 - creating 290
- Measure element 239
- memory 39
- Memory Data Sorter filter 39
- menus 129
- merge filters 45, 46, 48
- merging data rows 46, 48

- messages
 - See also* error messages
 - getting log 354
 - logging 343, 346, 354
 - outputting 349
- metadata
 - creating joins and 110
 - defining for ODA data sources 203
 - getting connections for 299
 - getting result set 280, 312, 330
 - returning from
 - ODA data sources 290, 294, 297
 - prepared statements 303
 - returning result set 280, 315
- metadata interfaces 294, 297, 303, 315
- methods
 - See also* functions
 - accessing data and 32
 - accessing stored procedures with 186
 - creating queries and 40, 41, 102
 - custom data drivers and 264
 - customizing connections and 28
 - customizing data streams and 31
 - filtering data rows and 33
 - ODA Java reference for 265
 - referencing names and 138
 - returning result sets with 190
 - synchronizing queries and 159
- Methods page (Properties) 85
- Microsoft Access databases 160
- Microsoft SQL databases connecting to 99
- multicolumn page layouts 240
- multiple cursor parameters 182
- multiple input filters 33, 45, 46, 50
- multiple input sources 29
- multiple result sets 181, 190
- Multiple-Input Filter setting 46
- multi-table queries 109, 110
- multi-table reports 46

N

- Name element
 - DataSource type 220
 - DataStreamDefinition type 241
 - InputFieldDefinition type 248
 - InputParameterDefinition type 251

- Name element (*continued*)
 - NameValuePair type 251
 - ODA connections and 219
 - OutputFieldDefinition type 255
 - Property type 256
 - PropertyType type 230
 - ResultColumn type 258
 - ResultSetDefinition type 204, 259
 - name qualification options (Database Browser) 106
 - names
 - See also* display names
 - accessing data sources and 91
 - adding space separators to 151
 - changing display 136
 - changing stored procedure 173
 - changing table 164
 - defaults for column and table 58
 - duplicating parameter 145
 - duplicating table 107
 - entering synonyms as 104
 - extracting IDs associated with 189
 - generating input parameter variable 176
 - getting column 316, 317
 - getting data source 300
 - getting driver 295
 - getting Logger 345
 - getting result set 281
 - qualifying 129
 - restrictions for 61
 - Names options (Database Browser) 105
 - name-value pairs 238, 251, 255
 - NameValuePair element 237
 - NameValuePair type 251
 - naming
 - ad hoc parameters 151
 - aggregate expressions 118
 - columns. *See* column names
 - computed fields 115
 - custom drivers 207
 - input parameters 176
 - ODA drivers 228
 - ODBC data sources 93
 - report parameters 145
 - result sets 204
 - static parameters 149
 - tables 107
 - native data types (Column Editor) 61
 - native data types. *See* data types
 - native database connection types 94
 - native database drivers 93, 96
 - NativeDataType element 226
 - NativeDataTypeCode element 226
 - NativeDataTypeType element 226
 - NativeTypeCode element
 - OutputFieldDefinition type 255
 - ResultColumn type 258
 - NativeTypeName element
 - OutputFieldDefinition type 255
 - ResultColumn type 258
 - NCHAR data types 180, 181
 - nested sections 55
 - network connections 4
 - NewConnection method 28
 - next method 314
 - non file-based databases 159
 - non-formatted data 54
 - non-sequential data access 32
 - non-String type parameters 282
 - non-supported operations 357
 - null values
 - determining if supported 306
 - specifying 315
 - testing for 286, 314, 318
 - numbers
 - getting decimal digits for 305, 306, 317, 318
 - getting values of 311
 - returning as doubles 279
 - returning as integers 279
 - setting values for 322, 323, 333
- ## O
- objects
 - constructing OdaException 357
 - deleting transient 29
 - getting data source 300
 - getting LogFormatter 270
 - obsolete tables 164
 - ObtainSelectStatement method 154, 156
 - ODA connection class 200, 215
 - ODA data source builder
 - accessing data sources for 236

- creating 200
- defined 202
- defining session state for 245
- defining session-specific locales for 260
- determining previous state 262
- returning definitions from 243
- setting response states for 256
- storing state of 204
- XML reference for 236
- ODA data source classes 201, 205
- ODA data sources
 - accessing 200, 201, 202, 236
 - aligning data in 246
 - changing properties for 204
 - closing connections for 289
 - committing changes to 290
 - connecting to 200, 203, 218, 289, 291
 - creating user interface for 200, 326
 - defining input parameters for 248
 - defining metadata for 297
 - defining output parameters for 255
 - defining result sets for 259
 - designing reports for 242, 260
 - determining columns in 238
 - determining parameter type for 301, 302
 - developing 219, 222, 228, 236
 - filtering columns in 244
 - getting collections of 300
 - getting interface version for 296
 - getting names 300
 - getting number of statements for 296
 - getting version of 299, 300
 - installing configuration files for 201
 - listing field definitions for 236, 237
 - listing parameter definitions for 236, 238
 - localizing 260
 - mapping data types for 207, 220, 221, 226, 252, 253
 - retrieving data from 203, 204
 - returning information about 356
 - returning result sets from 301, 302, 330
 - rolling back changes for 291
 - setting field definitions for 246, 254
 - setting properties for 203, 230, 240, 334
 - setting up data streams for 240
 - specifying multiple 219, 228
 - verifying connections for 291
- ODA data stream class 200, 215
- ODA data stream components 202
- ODA data streams 203
- ODA data types 207, 220, 221, 226, 252, 253
- ODA design tool
 - defining user locale for 260
 - displaying connections and 219, 221, 222
 - getting session information for 242, 244
 - retrieving data for 242
 - setting application locale for 243
- ODA driver libraries 223
- ODA drivers
 - accessing configurations for 206, 207
 - accessing data with 200, 236
 - accessing sample 200
 - adding configuration files for 207
 - backward compatibility for 228
 - building user interface for 202, 326
 - calling statements for 275
 - changing configurations for 201, 210
 - configuring 227
 - creating 200, 264
 - customizing properties for 222
 - debugging 202
 - defining connection information for 218, 294
 - defining operating system for 228
 - defining schemas for 210, 218
 - generating reports and 201, 205
 - getting metadata for 290
 - getting names 295
 - getting number of connections for 296
 - getting number of statements for 296
 - getting version 295
 - installing 200, 201, 207–208
 - loading libraries for 231
 - naming 207, 228
 - overview 200
 - retaining temporary files for 202
 - running 201
 - setting exceptions for 297, 326, 356, 357
 - setting logging configurations for 231, 293
 - specifying interface version for 207
 - specifying language for 224
- ODA Java reference 263, 265
- ODA metadata interfaces 294, 297, 303, 315
- oda package 205

- ODA run-time interface 230, 264, 296
- odaconfig.xml 207, 210, 218
- OdaDataType type 252
- OdaException class 206, 356
- OdaException objects 357
- ODAInterfaceVersion element 228
- OdaScalarDataType element 218, 226
- OdaScalarDataType type 226, 253
- ODBC administrator utility 90
- ODBC data sources
 - accessing 90
 - accessing data in 90, 99, 102
 - accessing stored procedures in 168
 - building queries for 102, 154
 - configuring 91
 - connecting to 8, 24, 90, 94, 95, 99
 - creating 90
 - defining run-time parameters for 146
 - filtering data in 144
 - limiting rows returned from 120
 - naming 93
 - restricting data retrieval from 144
 - retrieving data from 9, 90
 - selecting 93
 - setting paths for 160
 - specifying parameter type for 176
 - updating schemas for 161
 - verifying connections for 159
- ODBC DBMS module 181
- ODBC drivers 90, 92, 110
- ODBC Microsoft Access Setup dialog box 92
- odbcad32.exe 92
- OFF constant 339
- OFF_LEVEL constant 339
- Ok element 256
- OnColumnLayout element 240
- OnRead method 32, 37
- OnRow method 36
- open data access data drivers. *See* ODA data drivers
- open data access data sources. *See* ODA data sources
- open database connectivity. *See* ODBC
- open method 291
- OPEN_FAILURE constant 348
- OpenConnection method 28
- OpenCursor method 182
- OpenDataAccessConfig type 227
- opening
 - Column Editor 62, 65, 137
 - connections 28, 291
 - data adapters 32
 - Data Row Editor 59, 136
 - Database Browser 104
 - database cursors 32
 - Expression Builder 116, 123
 - input adapters 33
 - input sources 32
 - ODA driver libraries 231
 - Query Editor 104
 - Sample Parameter Values dialog 172
 - Scratch Pad 56
 - Stored Procedure Name Editor 173
 - text files 85
 - Textual Query Editor 128
 - XML files 32
- operating systems 225, 228
- Operational Data Store (ODS)
 - See also* ODS data sources; ODS objects
- operators
 - SQL queries 111
- optimizing
 - data retrieval 18
 - queries 102
- Oracle databases
 - See also* databases
 - accessing stored functions in 182
 - accessing stored procedures in 168, 180–186, 188, 193
 - connecting to 8, 94, 99, 159
 - creating queries for 103
 - displaying stored functions in 181
 - displaying stored procedures in 181
 - installing client software for 94
 - retrieving data from 9
 - returning result sets from 190–195
- Oracle DBMS module 181
- Oracle9i stored procedures 180
- ORDER BY clause 40, 139, 141
 - See also* SQL statements
- Order By page (Query Editor) 118, 140, 141
- OrderBy property
 - queries and 40
 - section components and 40

- sorting data and 41, 42, 139
- OSType element 225
- OSType type 228
- outer joins 110, 111
- output
 - changing fonts for 77
 - defining NCHAR data types and 181
 - flushing buffered 270, 367
- output files 269, 349
- output formats 258
- output parameters
 - See also* stored procedures
 - building ODA data sources and 203
 - checking for 178
 - creating 187, 255
 - defining NCHAR data types and 181
 - determining if null values in 286
 - determining if supported 302
 - finding 276
 - getting data source type 305
 - getting data type of 305, 331
 - getting date values in 278
 - getting decimal values in 277
 - getting number of 304
 - getting numeric precision for 305
 - getting numeric values in 279
 - getting string values in 282
 - getting structure values for 281
 - getting time values for 282
 - getting time values in 282
 - getting values of 187
 - listing definitions for 237, 238
 - overview 176
 - returning multiple result sets from 190, 192
 - returning specified 276
 - returning structure values for 281
 - returning time stamps for 283
 - setting decimal values for 332
 - setting field definitions for 254
 - specifying as type 176, 177
 - testing for 276
 - testing for null values in 286, 306
- output stream error constants 348
- output streams
 - closing 367
 - flushing 270, 367

- setting 368
 - writing log records to 366
- OutputFieldDefinition element 237
- OutputFieldDefinition type 254
- OutputParameterDefinition element 238
- OutputParameterDefinition type 255
- OutputParameters element 203, 241
- overriding methods 102
- overriding report connections 12
- owner information 105
- Owner pattern match setting 106

P

- page footers 68, 77
- page headers 68
- page layouts 54, 240
 - See also* designs
- page numbers 68, 77
- PageDecorationFont property 68, 77
- palettes. *See* Toolbox
- parallel sections 7, 55
- parameter definitions 177
- parameter mode constants 303, 304
- Parameter Properties dialog box 82
- Parameter Properties page 148, 151
- parameterized queries
 - See also* stored procedures
 - filtering data with 147
 - getting number of parameters in 304
 - viewing parameters for 134
- parameterModeIn constant 303
- parameterModeInOut constant 303
- parameterModeOut constant 303
- parameterModeUnknown constant 303
- parameterNoNulls constant 303
- parameterNullable constant 303
- parameterNullableUnknown constant 303
- parameters
 - accessing 206, 264
 - adding to expressions 146
 - assigning data types to 145, 148
 - assigning values to 134
 - binding to SQL statements 157
 - blank values and 134
 - changing data types for 149
 - changing properties for 148, 150, 151

- parameters (*continued*)
 - connecting to ODA data sources and 200, 236
 - connecting to XML files and 200
 - creating report 144, 149
 - defining NCHAR data types and 181
 - defining static. *See* static parameters
 - entering undefined 145, 177
 - filtering data and 144, 146
 - restrictions for 146
 - invalid values and 134
 - naming 145, 149, 151
 - prompting for 175
 - prompting for values with 145, 146
 - providing pick lists for 245
 - replacing 369
 - setting run-time 319
 - specifying default values for 146
 - specifying variables as 38, 82
 - stored procedures and 172, 176
 - viewing default values for 134
 - viewing properties for 148, 151
- Parameters command 148
- Parameters page (Stored Procedure Data Source Builder) 178
- passwords
 - data source server and 7, 94, 99
 - information objects and 76
 - ODBC data sources and 91, 99
- PATH variable 223
- paths
 - component libraries and 19
 - configuration files 14
 - connection configurations and 18
 - custom drivers and 207
 - data source drivers 160
 - executable files 223
 - file-based data sources 160
 - log files 232
 - ODA design tools 222
 - ODA drivers 208, 225
- PeopleSoft databases 168
 - See also* databases
- performance
 - graphical queries and 102, 107, 156
 - published components and 19
 - retrieving data and 9
 - shared connections and 7
 - sorting and 39
 - stored procedures and 168
 - textual queries and 102
- performance statistics 134
- pick lists 245
- plug-ins 200
- Position element 259
- precision (numeric values) 305, 317
- predefined connections 8, 12, 24
- predefined data drivers 200
- predefined data sources 24, 200, 210
- predefined databases 24
- Prepare method 99, 191
- prepare method 332
- Presorted setting 41, 141, 142
- presorting data 40, 41, 141
- Preview Column Data button 114
- Preview Data command 130, 133
- Preview Data dialog box 107, 114
- Preview Query Data dialog box 134
- Preview Table Data command 107
- previewing data 107, 114, 154
- previewing result sets 130, 133
- previous method 321
- primary keys 110
 - See also* joins
- primary result sets 241
- PrimaryResultSet element 203, 241
- private properties 241
- PrivateConnectionProperties element
 - DesignSessionRequest type 204, 243
 - DesignSessionResponse type 204, 244
- PrivateProperties element 203, 241
- privileges 73
- procedures 181
 - See also* methods
- ProductName element 232
- programming environment 28
- Progress databases 8, 9
 - See also* databases
- prompting for input 145, 146
- prompting for parameters 175
- prompting for query expressions 149
- prompting for user input 145
- prompting for user-specified values 146

- properties
 - See also* Properties page
 - ad hoc parameters 150, 151
 - custom data source builder and 200
 - custom drivers and 200, 236
 - data row columns 62, 138
 - data source server connections 9
 - database connections 94, 96
 - fonts 67, 68, 77
 - information objects 67
 - name-value pairs for 238, 255
 - native data sources 236
 - ODA data sources 203, 204, 230, 240, 334
 - ODA data streams 240
 - ODA statement objects 335
 - ODBC connections 9
 - run-time parameters 145
 - setting data source 20, 23
 - static parameters 148
 - stored procedures 169
 - table 109
 - textual queries and 131
 - Properties command 96
 - Properties element
 - Connection type 219
 - DataSource type 220
 - Properties page
 - accessing 56
 - changing font settings on 67, 68, 78
 - creating custom data sources and 38
 - database connections and 24
 - setting database connections and 96
 - setting information object properties 76
 - setting sorting options from 41, 42
 - Properties type 229
 - Property attribute (configurations) 20
 - Property element 225, 229, 238
 - property lists 221, 225
 - property sheets. *See* Properties page
 - Property type 255
 - PropertyType type 230
 - ProviderError element 256
 - public instance variables 48, 50
 - public properties 241
 - PublicConnectionProperties element
 - DesignSessionRequest type 203, 204, 243
 - DesignSessionResponse type 203, 204, 244
 - PublicProperties element 203, 241
 - publish method
 - FileHandler 267
 - Handler 271
 - StreamHandler 367
- ## Q
- QBE expressions 120, 122
 - entering at run time 149
 - QBE syntax 122
 - queries
 - See also* graphical queries; parameterized queries; textual queries
 - building for
 - custom data sources 38
 - databases 168
 - ODA data sources 201, 202, 203
 - changing result sets for 135–138
 - converting 131–133, 156
 - creating 98, 102, 103, 128, 290
 - customizing 112, 157
 - defining data streams for 240
 - developing user interface for 210
 - displaying performance statistics for 134
 - entering statements for. *See* SQL statements
 - executing 157
 - filtering data with 132, 147, 150
 - formatting output for 77–78
 - getting state 301
 - installing custom drivers and 200
 - instantiating data rows for 34
 - joining tables for 109, 110
 - maintaining data source information for 154, 156
 - merging data with 46
 - naming parameters for 149, 151
 - optimizing 102
 - overview 102
 - prerequisites for 97
 - presorting data for 40, 141
 - previewing result sets for 133–135, 154
 - prompting for user input and 145, 146, 149
 - replacing automatically generated 112
 - restricting data retrieval for 120, 124, 144
 - retrieving data with 90, 97, 128

queries (*continued*)

- returning aggregate rows with 124
 - returning computed fields with 37, 115, 116
 - returning multiple results sets 168
 - returning summary data with 117–120
 - running 157, 329, 334, 335
 - sorting data with 39, 40, 118, 139–142
 - synchronizing 102, 156, 166
 - tracking changes to 164
 - updating 161, 163
 - verifying connections for 159, 160
- Query by Example. *See* QBE expressions; QBE syntax
- query data source components 96, 97
- query data sources
- accessing databases and 99
 - accessing ODBC data sources and 99
 - adding 96–98
 - creating for textual queries 128
 - retrieving multiple tables and 47
- query data streams 96
- Query Editor
- See also* graphical queries; queries
- adding tables to 106
 - changing column order with 116
 - changing data types with 116
 - changing join operator for 111
 - changing SQL statements with 113
 - changing table properties with 109
 - creating aggregate rows with 117
 - creating computed fields with 115, 116
 - creating joins with 110
 - creating queries with 34
 - defining ad hoc parameters with 150
 - defining user-specified conditions with 146
 - deleting joins from 112
 - determining synchronization status 162
 - displaying data sources and owner information for 105
 - displaying query statements 127, 154
 - freezing columns and rows in 108
 - opening 104
 - overview 102, 104
 - refreshing data dictionary for 161
 - removing columns with 114, 119

- removing conditions with 123, 124, 125, 126
- removing tables from 109
- resizing columns for 107
- selecting columns with 113
- setting conditions with 123, 125, 126
- setting Database Browser options for 105
- sorting with 140
- summarizing data with 118
- verifying connection properties 160

QueryText element 245

question mark (?) character 151

quotation mark characters. *See* double quotation mark character; single quotation mark character

R

radio buttons 250

random data access 32

range of values 122

read privilege 73

read-only queries 40, 41

ReadOnly value 260

reconfiguring ODA drivers 201, 210

RecordDefinition element

 InputParameterDefinition type 251

 OutputParameterDefinition type 255

records

See also rows

copying values from 34

creating joins for 110

determining logging status for 271

filtering 43

formatting log file 341, 360

getting formatter for 270

getting logging level for 270

grouping 119

matching from multiple tables 109

outputting to specific destination 269

publishing to log files 267, 268, 271

retrieving 32, 99

setting formatter for 272

sorting 39, 118

REFCUR3.GETMGRDATA function 182

Reference names field 138

- references
 - column names and 103, 138
 - component libraries and 24
 - report components and 30
 - SQL queries and 138, 159
 - SQL query conversions and 133
 - table names and 109
- Refresh Data Dictionary setting 161
- refreshing configuration files 13
- refreshing data dictionaries 161
- regions. *See* frames
- relational databases. *See* databases
- relational expressions 122
- relative paths 18
- removing. *See* deleting
- renaming
 - stored procedures 173
 - tables 107, 164
- report components. *See* components
- report design components 28
- report designs
 - accessing
 - ODA data sources and 201, 204, 240
 - accessing stored procedures for 169–171, 175
 - adding
 - connections to 19, 22
 - building computed values for 36
 - creating xv
 - creating connections in 7–9, 95
 - creating SQL queries and 97
 - defining alternative data types for 220, 226
 - developing 28
 - including configurations in 17
 - installing custom drivers for 200, 206
 - migrating to production environments 23
 - referencing components in 30
 - referencing data sources in 159
 - removing default configurations for 14
 - retrieving data for 9
 - sample configurations for 15
 - saving 38
 - selecting data sources for 24, 84, 200, 210
 - viewing data row variables in 58, 59
- Report Encyclopedia. *See* Encyclopedia volumes
- report executables. *See* report object executable files
- report files 54, 159, 200, 222
 - See also* specific type
- report object design files
 - changing fonts for 67, 77
 - copying 159
 - displaying data row variables in 58
 - displaying data rows in 55, 58
 - generating executable files from 55
 - updating query information in 158
- report object executable files 7, 114
 - See also* executable files
- report object library files 20, 158, 159, 200
 - See also* libraries
- report parameters 241
 - See also* parameters
- report sections
 - adding to designs 55
 - defining multiple data sources for 45
 - displaying data rows in 54, 56
 - placing connections in 5, 7, 28
 - presorting data for 41
 - retrieving data for 6, 7, 30
 - sorting data in 39, 40, 41, 103
 - viewing nested 55
- report servers. *See* iServer; servers
- Report Structure window 30
- Report Viewer 56
- reportError method 271
- reports
 - accessing ODA data sources for 200
 - adding titles to 144
 - changing default data access for 28
 - changing fonts for 67, 68
 - creating for
 - custom data sources 39
 - data from multiple sources 45
 - information objects 73
 - ODA data sources and 242, 260
 - stored procedure data sources 169, 175, 186
 - text file data sources 80, 84
 - customizing 28
 - designing. *See* designing reports; designs
 - generating 4
 - installing custom drivers for 201, 205

- reports (*continued*)
 - laying out 54
 - overview 4
 - presorting data for 41
 - running 201
 - selecting data for 9, 32, 120
- ReportType property 54, 55, 56, 58
- request and response streams 236, 242, 243
- request-and-response formats 203, 206, 236
- Requester dialog box 86, 145, 149
- Requester page (Information Console) 122
- required connections 4
- required data streams 169
- required parameters 134
- resizing columns 61, 103, 107, 138
- resources 14
- response states (ODA) 256
- ResponseState element 204, 244
- ResponseState type 256
- Result Columns page 173, 175
- result set interface 307
- result sets
 - accessing 264
 - changing 135–138
 - closing cursors in 308
 - defining columns for 130, 238, 256, 259
 - defining metadata for 315
 - defining multidimensional attributes for 239
 - determining column axis type in 239
 - getting column names for 363
 - getting columns in 308, 316, 317
 - getting current 331
 - getting metadata for 280, 312, 330
 - getting names of 281
 - getting number of rows returnable 329
 - getting sort order for 281
 - getting specified 280
 - mapping data types to 135, 138
 - moving cursors in 307, 314, 330
 - naming 204
 - previewing 130, 133
 - renaming stored procedures and 174
 - returning from
 - ODA data sources 203, 204, 206
 - Oracle data sources 180, 181
 - stored functions 181
 - stored procedures 173, 178, 187
 - returning multiple 181, 190–195, 301, 302, 330
 - running queries for 99, 241, 329, 334, 335
 - setting new rows for 284
 - setting number of rows returned 314, 334
 - sorting data in 285, 361
 - sorting from Textual Query Editor 141
 - specifying as primary 241
 - specifying new rows for 284
 - specifying tables for 284
 - summarizing data from 117
 - testing for null values in 314
 - validating queries for 156, 159
 - viewing default parameters for 134
 - viewing duplicate columns in 173
- ResultColumn element 238
- ResultColumn type 256
- ResultSetColumns element 204, 259
- ResultSetDefinition element 203, 204
- ResultSetDefinition type 259
- return values 181, 188
- Rewind method 33
- RFM data sources
 - customizing data streams for 32
- RFM export parameters 32
- Right element 246
- right outer joins 110
- .rod files. *See* report object design files
- .rol files. *See* report object library files
- rollback method 291
- rolling back transactions 291
- row numbers 48, 50
- row sets 319
- rows
 - See also* data row components; data row objects
 - adding columns to 64
 - appending to collections 319, 320
 - binding columns to 102
 - building aggregate 117–120
 - changing fonts for 68
 - creating 34
 - defining custom data streams for 32
 - displaying 54, 55, 56, 58
 - excluding duplicate 114
 - fetching with cursor variables 193

- filtering 33, 43
- getting number returned 329
- getting position of 32, 312
- getting structures for 281
- grouping 117
- merging 46, 48
- moving cursors for 307, 314, 320, 321
- overview 34
- previewing 108, 133
- processing sequentially 50
- removing columns from 66
- removing conditions on 123, 124
- retrieving with conditions 120, 122, 123
- returning from
 - custom data sources 38
 - data adapters 45
 - multiple data sources 45, 46, 48
 - stored procedures 187
 - text file data sources 83
- selecting data for 64, 120
- setting formats for 61
- setting number returned 314, 334
- setting return values for 35, 36
- sorting 39, 41, 42, 140
- specifying for input parameters 284
- storing computed values in 37
- .rox files. *See* report object executable files
- running queries 157, 329, 334, 335
- running reports 201
- running stored procedures 187
- run-time connections 22
- Runtime element (configurations) 23
- run-time filters
 - constructing expressions for 149
 - defining parameters for 144, 149
- run-time interface (ODA) 230, 264, 296
- run-time parameters 319
- run-time queries 123, 145, 149
- run-time values 145
- RunTimeInterface type 230

S

- sample configuration file 13, 15
- sample databases 91
- sample ODA driver 200
- sample ODBC data sources 91

- Sample Parameter Values command 172
- Sample Parameter Values dialog box 172, 178, 180
- sample values 172, 178
- sample_configuration_file.xml 15
- saving
 - components 55
 - report designs 38
 - temporary files 202
- scalar data types 218, 226, 253
- scalar parameters
 - ODA data sources and 245, 276
- schemas
 - accessing data and 90
 - configuring ODA custom drivers and 210, 215
 - creating ODA data sources and 236
 - determining if current 162
 - updating 158, 161
- Scratch Pad 55, 56
- search expressions 174
- search paths 18
- searching for
 - embedded strings 369
 - stored procedures 174
- section components 6
- sections
 - See also* specific type
 - adding to designs 55
 - defining multiple data sources for 45
 - displaying data rows in 54, 56
 - placing connections in 5, 7, 28
 - presorting data for 41
 - retrieving data for 6, 7, 30
 - sorting data in 39, 40, 41, 103
 - viewing nested 55
- SeekBy method 33
- SeekTo method 33
- SeekToEnd method 33
- select all operator 130
- Select Component dialog box 8, 95
- Select Database dialog box 93
- select painter. *See* Query Editor
- SELECT statements
 - See also* SQL statements
 - automatically generating 102, 127
 - building manually 102, 128

- SELECT statements (*continued*)
 - creating 128
 - database cursors and 99
 - Describe Query operations and 130, 137
 - naming parameters for 149, 151
 - nesting 124
 - obtaining 154, 156
 - OrderBy property and 41
 - parameter names in 151
- selection formulas. *See* parameters
- SelectNameValueList element 251
- SelectValueList element
 - InputFieldDefinition type 248
 - InputParameterDefinition type 251
 - PropertyType type 230
- sequential sections 7, 55
- server names 7
- servers
 - See also* iServer
 - connecting to 7, 9, 94, 96
 - running textual queries and 130
- session state 245, 262
- SessionEditMode element 243
- SessionEditMode type 260
- SessionLocale element 204, 243
- SessionLocale type 260
- setBigDecimal method
 - IResultSet 321
 - IStatement 332
- setDate method
 - IResultSet 321
 - IStatement 332
- setDouble method
 - IResultSet 322
 - IStatement 333
- setFilter method 272
- setFormatter method
 - Handler 272
 - StreamHandler 368
- setHandler method 346
- setInt method
 - IResultSet 323
 - IStatement 333
- SetJavaThreadContextClassLoader
 - element 224
- setLevel method
 - Handler 272
 - Logger 346
 - LogRecord 354
- setLogConfiguration method 206, 292
- setLoggingErrorHandler method 272
- setMaxRows method
 - IResultSet 314
 - IStatement 334
- setMessage method 354
- setMillis method 355
- setNewRow method 283
- setNewRowSet method 284
- setNextException method 359
- setOutputStream method 368
- setProperty method 334
- setPropertyInfo method 335
- setSortSpec method
 - ICallStatement 285
 - IStatement 335
- setString method
 - IResultSet 323
 - IStatement 335
- setThrown method 355
- setTime method
 - IResultSet 324
 - IStatement 336
- setTimestamp method
 - IResultSet 325
 - IStatement 337
- settings. *See* properties
- SetupDataRow method 34
- SetValue method 35, 36
- SEVERE constant 339
- SEVERE level messages 347
- severe method 347
- SEVERE_LEVEL constant 339
- sharing connections 7
- ShortDescription element 203, 242
- Show system objects setting 106
- SimpleFormatter class 206, 360
- Single data type 187
- Single Input Filter setting 44
- single input filters 33
- single value expressions 123
- size method 325
- sort criteria 40, 139, 362
- sort filter utility 141

- sort filters
 - adding for custom data sources 39
 - creating 42, 141
 - restrictions for 33, 139
- sort keys
 - adding 362
 - associating with group sections 40, 139, 141
 - associating with statements 361
 - caution for 40, 141
 - counting 364
 - defined 39
 - getting column names for 363
- sort mode constants 297
- sort modes 301, 361, 362, 364
- sort options 141
- sort order
 - changing programmatically 43
 - defining for custom data sources 40, 41
 - determining default 103
 - getting 281, 364
 - setting 361
 - specifying with queries 39, 140
- sort order constants 361, 364
- sort specifications (ODA) 281, 285, 332, 335
 - See also* SortSpec class
- sortAsc constant 361, 364
- sortDesc constant 361, 364
- sorting criteria 39, 40
- sorting data
 - custom data sources and 32, 38, 39, 40
 - databases and 139
 - defaults for 39, 40, 141
 - from report designs 42
 - guidelines for 39
 - in result sets 281, 361
 - non-sequential data streams and 32
 - ODA result sets and 281, 285, 361
 - overview 39
 - with OrderBy property 41, 42, 139
 - with queries 118, 139–142
- sorting information, preserving 55
- Sorting page 40
- sorting precedence 40, 139
- Sorting property 39, 41, 142
- sortModeColumnOrder constant 297, 301, 362, 364
- sortModeNone constant 297, 301, 362, 364
- sortModeSingleColumn constant 297, 301, 362, 364
- sortModeSingleOrder constant 297, 301, 362, 364
- SortSpec class 206, 361
- source code
 - See also* Actuate Basic
 - accessing text files and 5, 85
 - creating locale-specific reports and 243
 - customizing data streams and 31
 - executing stored procedures and 190
 - filtering data and 43, 46
 - implementing custom drivers with 200
 - referencing column names in 103, 138
 - sorting data and 40
 - throwing errors and 356
 - verifying input values with 150
- special characters
 - See also* specific character
- specialized reporting 144
- Specify Parameter Values dialog box 134
- spreadsheet files 32
- spreadsheet server. *See* iServer
- SQL data sources. *See* query data sources
- SQL databases
 - See also* databases
 - connecting to 8
 - retrieving data from 9, 32, 34
- SQL expressions 122, 123, 126
- SQL page (Query Editor) 127, 154
- SQL Server databases
 - connecting to 94
- SQL statements
 - See also* queries
 - adding conditions to 147, 150, 151
 - adding functions to 116, 118
 - adding sort keys to 141
 - applying conditions with 120, 122, 123, 124, 126
 - binding parameters to 157
 - building automatically 102, 104
 - building manually 102
 - building text-based. *See* textual queries

- SQL statements (*continued*)
 - copying 128
 - creating 290
 - displaying 126, 127, 154
 - entering at run time 123, 145, 149
 - overriding 155, 156
 - referencing columns in 165
 - referencing tables or views in 109
 - removing column references in 165
 - removing columns from 114, 119
 - removing conditions from 124, 126
 - removing tables from 109
 - reordering columns in 116
 - select all operator in 130
 - selecting columns for 113, 116, 117
 - selecting predefined data sources for 200
 - selecting tables for 104, 106, 109
 - updating 156, 157, 158, 166
- SQLSTATE value 301, 356, 359
- sqlStateSQL99 constant 297
- sqlStateXOpen constant 298
- Start method
 - building custom data streams and 32
 - creating input adapters and 48
 - creating input adapters with 51
 - displaying CSV data and 84, 85
 - filtering data and 33
 - generating content with 35
 - instantiating connections and 28
- StartNextSet method 187
- state 245, 262
- StateInfo element 246
- StateInfo type 262
- StateInfoBlob element 262
- StateInfoString element 262
- statement interface 326
- statements
 - accessing 206, 264
 - associating sort keys with 361
 - calling 275
 - closing 328
 - creating 290
 - defined 326
 - determining parameter type for 276, 303
 - determining supported parameters for 301, 302
 - executing 328, 334, 335
 - getting current result set 331
 - getting metadata for 330
 - getting number of active 296
 - getting number of rows returned 329
 - getting parameters for 304, 330, 331
 - getting result sets for 281
 - getting sort specifications 332
 - preparing 332
 - returning number of parameters in 304
 - returning result sets from 301, 302
 - setting number of rows returned 334
 - setting properties for 335
 - specifying sort specification 335
- static parameters
 - adding to queries 113, 123, 147, 148
 - changing data types for 149
 - prompting for input and 145, 146
 - restrictions for 181
 - setting properties for 148
 - viewing properties for 148
- Static Parameters page (Textual Query Editor) 148, 149
- static variables 82
- stored function icon 181
- stored functions 180, 181, 182, 190
- Stored Procedure Browser 171, 174, 181
- Stored Procedure command 174
- stored procedure controls 175
- Stored Procedure Data Source Builder
 - accessing Sample Parameter Values dialog from 172
 - adding weak cursor type definitions with 180
 - checking for required parameters with 178
 - designating parameter type with 176, 177
 - opening 171
 - returning multiple result sets and 181
 - selecting stored procedures for 173, 174
 - specifying parameter type in 176
- stored procedure data sources 169, 171
 - See also* stored procedures
- Stored Procedure Name Editor 173
- Stored Procedure Name Editor
 - command 173
- Stored Procedure page (Options) 174

- stored procedures
 - See also* input parameters; output parameters
 - accessing definitions for 190
 - accessing multiple result sets for 190–195
 - calling statements for 275
 - checking for parameters in 178
 - creating cursors for 187
 - creating custom data streams for 32, 186
 - customizing 186–195
 - designing for 169–171
 - displaying 171, 174
 - entering parameters for 175, 176
 - entering sample values for 172, 178
 - executing on Oracle databases 180, 188, 190
 - executing programmatically 187
 - extracting IDs from 189
 - extracting return values from 188
 - filtering data with 43
 - overview 168
 - placing data in reports from 175–176
 - renaming 173
 - retrieving data with 90, 128, 175
 - returning results sets for 168, 173
 - returning status values with 187
 - searching for 174
 - selecting 171, 174
 - setting sort order for 40, 41
 - specifying parameter type for 176, 177
 - synchronizing 168, 180
 - verifying contents 180
- StoredProcedureSource data stream 169, 170
- StreamedResultSetBufferSize element 231
- StreamHandler class 206, 366
- streaming log records 366
- String data type 187, 323, 335
- String element
 - ArrayOfString type 218, 239
 - OdaDataType type 253
 - OdaScalarDataType type 254
- string substitution 369, 372, 373
- strings
 - assigning to parameters 335
 - comparing 103
 - expressions and 116, 118
 - formatting 341, 360
 - getting column values as 312
 - getting delimited 369
 - getting output parameter 282
 - getting version numbers as 295
 - locating embedded 369
 - returning sort specification as 365
 - setting values for 323
- StringSubstitutionUtil class 206, 369
- Structure element 253
- structure parameters
 - ODA data sources and 276, 281
- structures 319
- subclassing components 24
- substituteByIndex method 372
- substituteByName method 373
- substrings 287
- summarizing data 116, 117–120
- summary data 117, 118
- supportsInParameters method 301
- supportsMultipleOpenResults method 301
- supportsMultipleResultSets method 302
- supportsNamedParameters method 302
- supportsNamedResultSets method 302
- supportsOutParameters method 302
- synchronization tools 158
- Synchronize Query button 162
- Synchronize Query command 166
- Synchronize Query with Schema dialog
 - box 162, 163
- Synchronize Stored Procedure command 180
- synchronizing
 - queries 102, 156, 166
 - stored procedures 168, 180
- synonyms 104, 106
- system DSNs 91
- system tables 104

T

- Table element 253
- "Table invalid or not found" message 164
- table names
 - changing 109
 - entering aliases for 107
 - formatting 109
 - including data source information in 105
 - qualifying 129

- table names (*continued*)
 - returning truncated 54
- Table or view pattern match setting 106
- Table Property page (Query Editor) 109, 160, 164
- tables
 - See also* databases
 - adding to queries 106, 109
 - building at run time 284, 319
 - changing properties for 109
 - checking database connections for 160
 - choosing columns in 113, 129
 - creating joins for 46, 109, 110
 - input parameters and 284
 - linking related 109
 - matching records in multiple 109
 - naming 107
 - referencing 109
 - referencing columns in 133, 138
 - removing from queries 109
 - removing joins from 109, 112
 - renaming 107, 164
 - retrieving data for 32
 - selecting 104
 - specifying fields in 18
 - synchronizing queries and 102
 - updating 163
 - viewing columns in 108
 - viewing data in 107, 114, 154
- Tables and Views options (Database Browser) 105
- Temp directory 202
- TEMP environment variable 202
- temporary files 202
- temporary objects 29
- testing
 - for invalid values 314
 - for null values 286, 314, 318
- text boxes 248, 250
- text editors 15
- text file data sources. *See* text files
- text files
 - accessing data in 80
 - closing 85
 - connecting to 5, 81
 - customizing data sources for 80, 81
 - customizing data streams for 32
 - defining data row variables for 83
 - designing for 84
 - displaying data from 86
 - opening 85
 - retrieving data from 80, 85
 - specifying connection properties in 12
- text formats 259
- text strings. *See* strings
- TextFormat element 259
- textual queries
 - adding parameters to 148–149, 151
 - applying conditions with 147, 151
 - case-insensitivity in 103
 - changing 129, 131
 - changing result sets for 135, 137
 - changing statements for 155
 - clearing 130
 - converting graphical queries to 97, 131–133, 156
 - creating 98, 102, 128–131
 - displaying statements for 154
 - filtering data with 103, 132, 150
 - overriding statements for 155
 - previewing 130, 133, 154
 - qualifying table names for 129
 - selecting columns for 129
 - setting properties for 131
 - sorting data with 141, 142
 - synchronizing 156, 157
 - tracking changes to 164
 - trimming trailing spaces for 103
 - undoing changes to 130
 - updating 157
- textual query data sources 128
 - See also* query data sources
- Textual Query Editor
 - accessing 128
 - changing properties with 131
 - defining ad hoc parameters with 151, 152
 - defining static parameters with 148, 149
 - displaying context menu for 129
 - displaying SQL statements 154
 - entering SQL statements with 34, 129
 - erasing statements in 130
 - overview 102
 - restrictions for 138
 - sorting with 141

- updating information for 130
- third-party databases 176
- time
 - See also* time stamps
 - getting 282, 313, 354
 - setting 324, 336, 355
- Time data type 324, 336
- Time element
 - OdaDataType type 253
 - OdaScalarDataType type 254
- time stamps
 - getting 283, 313
 - setting 325, 337
- Timestamp data type 325, 337
- Timestamp element
 - OdaDataType type 253
 - OdaScalarDataType type 254
- TitleFont property 68, 77
- titles 68, 77
 - adding to reports 144
- Toolbox 6, 97
- toString method 365
- totals 175
- TraceLogging element 231
- TraceLogging type 231
- trailing spaces 103
- transactions
 - committing 290
 - rolling back 291
- transient files. *See* temporary files
- transient objects 29
- TRIM function 103
- truncated characters 61
- truncated columns 107, 135
- truncated table and column names 54
- Type attribute (configurations) 19, 22
- Type element 203, 242
- Type of Join settings 111
- types. *See* data types

U

- undefined parameters 145, 177
- unformatted data 54, 55
- union filters 45, 50
- unique values 111
- UNKNOWN keyword 176

- unknown parameters 176
- unnamed delimited strings 369, 372, 373
- UnsupportedOperationException
 - constant 264
- UnsupportedOperationException objects 357
- UPDATE statements 157
- updating
 - configurations 201
 - data dictionaries 161
 - data sources 156
 - database schemas 158, 161
 - queries 161, 163
 - SQL statements 156, 157, 158, 166
 - stored procedures 180
- UPPER function 103
- URLs
 - information objects and 76
 - specifying search paths with 18
- user DSNs 91
- user interfaces 200, 202
- user names
 - accessing databases and 94, 99
 - accessing information objects and 76
 - accessing ODBC data sources and 91
 - creating connections and 7
- UserCancelled element 256
- users
 - entering filters at run time 144, 149
 - prompting for input 149
 - prompting for input from 145
 - prompting for specific values 145, 146, 150

V

- V_CURRENCY type code 187
- V_DATE type code 187
- V_DOUBLE type code 187
- V_INTEGER type code 187
- V_LONG type code 187
- V_SINGLE type code 187
- V_STRING type code 187
- Value element
 - NameValuePair type 251
 - Property type 256
 - PropertyType type 230
- value expressions 123
- ValueColumn element 245

- ValueExp property 35
 - See also* value expressions
- values
 - See also* data
 - assigning to parameters 134, 146
 - comparing 43, 103
 - computing 36, 115
 - displaying default 134
 - entering invalid 134
 - entering null 315
 - getting decimal precision for 305, 317
 - inputting sample 172, 178
 - prompting for 145, 146, 150
 - QBE conventions for 122
 - returning unique 111
 - selecting at run time 145, 149
 - selecting data and 120
 - setting date 321, 332
 - setting decimal 321, 332
 - setting for controls 36
 - setting numeric 322, 323, 333
 - setting string 323, 335
 - setting time 324, 336, 337
 - setting time stamp 325
 - testing for invalid 314
 - testing for null 286, 314, 318
 - verifying user-specified 150
- variable names 61
- variable type mappings 187
- variables
 - See also* data row variables
 - accessing multiple data sources and 50
 - associating with data rows 34, 61
 - binding to columns 187
 - changing 58, 60
 - connecting to text files and 81
 - creating computed fields and 37, 113, 115
 - defining data sources and 38
 - defining static 82
 - displaying 59, 84, 138
 - merging data rows and 48
 - naming parameters and 176
 - redefining attributes of 60
 - specifying parameters as 82, 177
- Variables page (Properties) 81
- Variant data types 116, 187
- Vendor element 232, 233
- VendorInfo type 232
- VendorPhone element 233
- VendorURL element 233
- verifying stored procedure content 180
- Version element 232, 246
- version numbers
 - getting custom driver 295
 - getting data source 299, 300
 - getting ODA interface 296
- Viewer 56
- viewing
 - column names 165
 - connection properties 9, 169
 - data 54, 107, 114
 - data row variables 59, 138
 - data rows 54, 55, 56, 58
 - default values 134
 - nested sections 55
 - ODBC drivers 92
 - owner information 105
 - performance statistics 134
 - query statements 127
 - sample configurations 15
 - SQL statements 126, 154
 - stored functions 181
 - stored procedures 171, 174
 - system data sources 92
 - unformatted data 55
- views
 - displaying 104
 - formatting names 109
 - selecting 104, 106
 - updating 163
- Visual Basic type codes 187
- volumes. *See* Encyclopedia volumes

W

- WARNING constant 339
- WARNING level messages 347
- warning method 347
- WARNING_LEVEL constant 339
- WarnValueFormatAdjustment element 223
- wasNull method
 - ICallStatement 285
 - IRResultSet 314
- weak cursors (defined) 180

WHERE clause 109, 120, 122

See also SQL statements

setting at run time 150

white space characters

in columns 103

in queries 151

wizards

See also specific report wizard

accessing information objects with 73

creating queries and 97

defining default connections for 19

retrieving data with 9

specifying data sources for 24

specifying sort order with 40

word wrapping 259

Wrap element 259

WRITE_FAILURE constant 348

X

XML data sources 8, 9, 32

XML documents 12, 15

XML elements 218, 236

XML files

ODA data source builder and 200, 202, 236

opening 32

XML request and response streams 236, 242,
243

XML schemas 236

