

Developing Database Applications

JBuilder® 2005

Borland®
Excellence Endures™

Borland Software Corporation
100 Enterprise Way
Scotts Valley, California 95066-3249
www.borland.com

Refer to the file `deploy.html` located in the `redist` directory of your JBuilder product for a complete list of files that you can distribute in accordance with the JBuilder License Statement and Limited Warranty.

Borland Software Corporation may have patents and/or pending patent applications covering subject matter in this document. Please refer to the product CD or the About dialog box for the list of applicable patents. The furnishing of this document does not give you any license to these patents.

COPYRIGHT © 1997–2004 Borland Software Corporation. All rights reserved. All Borland brand and product names are trademarks or registered trademarks of Borland Software Corporation in the United States and other countries. All other marks are the property of their respective owners.

For third-party conditions and disclaimers, see the Release Notes on your JBuilder product CD.

Printed in the U.S.A.

JB2005database 10E13R0804
0405060708-9 8 7 6 5 4 3 2 1
PDF

Contents

Chapter 1		Chapter 4	
Introduction	1	Connecting to a database	27
Chapter summaries	2	Connecting to databases	28
Database tutorials	3	Adding a Database component to your	
Database samples	3	application	28
Related documentation	4	Setting Database connection properties	29
Documentation conventions	6	Setting up JDataStore	31
Developer support and resources	7	Setting up InterBase and InterClient	31
Contacting Borland Developer Support	7	Using InterBase and InterClient with JBuilder	32
Online resources	7	Tips on using sample InterBase tables	32
World Wide Web	8	Adding a JDBC driver to JBuilder	33
Borland newsgroups	8	Creating the .library and .config files	33
Usenet newsgroups	8	Adding the JDBC driver to projects	34
Reporting bugs	8	Connecting to a database using InterClient	
		JDBC drivers	35
Chapter 2		Using the Database component in your	
Understanding JBuilder database		application	37
 applications	11	Prompting for user name and password	38
Database application architecture	11	Pooling JDBC connections	38
DataExpress components	12	Optimizing performance of JConnectionPool	40
Key features and benefits	13	Logging output	40
Overview of the DataExpress components	14	Pooling example	40
DataExpress for EJB components	17	Troubleshooting JDataStore and InterBase	
InternetBeans Express	17	connections	43
XML database components	17	Common connection error messages	43
dbSwing	17		
Data modules and the Data Modeler	18	Chapter 5	
Database Pilot	18	Retrieving data from a data source	45
JDBC Monitor	19	Querying a database	46
JDataStore and JBuilder	19	Setting properties in the query dialog box	47
When to use JDataStore versus JDBC drivers	19	The Query page	47
Additional advantages of a JDataStore	20	The Parameters page	48
Using the JDataStore Explorer	20	Place SQL text in resource bundle	49
JDataStore explorer operations	21	Querying a database: Hints & tips	51
InterBase and JBuilder	21	Enhancing data set performance	51
		Persisting query metadata	52
Chapter 3		Opening and closing data sets	52
Importing and exporting data		Ensuring that a query is updatable	52
 from a text file	23	Using parameterized queries to obtain data	
Adding columns to a TableDataSet in the editor	23	from your database	53
Importing formatted data from a text file	24	Parameterizing a query	53
Retrieving data from a JDBC data source	24	Creating the application	53
Exporting data	25	Adding a Parameter Row	54
Exporting data from a QueryDataSet to a		Adding a QueryDataSet	54
text file	25	Add the UI components	55
Saving changes from a TableDataSet to a		Parameterized queries: Hints & tips	57
SQL table	25	Using parameters	57
Saving changes loaded from a		Re-executing the parameterized query	
TextDataFile to a JDBC data source	26	with new parameters	59
		Parameterized queries in master-detail	
		relationships	59

Chapter 6

Using stored procedures 61

Stored procedures: hints & tips	62
Escape sequences, SQL statements, and server-specific procedure calls	62
Using vendor-specific stored procedures	63
Using JDataStore stored procedures and user-defined functions	63
Using InterBase stored procedures	63
Using parameters with Oracle PL/SQL stored procedures	64
Using Sybase stored procedures	65
Sample application with database-server specific stored procedures	65
Writing a custom data provider	65
Obtaining metadata	66
Invoking initData	66
Obtaining actual data	67
Tips on designing a custom data provider	67
Understanding the provideData() method in master-detail data sets	67

Chapter 7

Working with columns 69

Understanding Column properties and metadata	69
Non-metadata Column properties	69
Viewing column information in the column designer	70
Generate RowIterator Class button	71
Using the column designer to persist metadata	71
Making metadata dynamic using the column designer	72
Viewing column information in the Database Pilot	72
Optimizing a query	73
Setting column properties	73
Setting Column properties using JBuilder's visual design tools	73
Setting properties in code	73
Persistent columns	73
Combining live metadata with persistent columns	74
Removing persistent columns	74
Using persistent columns to add empty columns to a DataSet	75
Controlling column order in a DataSet	75

Chapter 8

Saving changes back to your data source 77

Saving changes from a QueryDataSet	78
Adding a button to save changes from a QueryDataSet	79
Saving changes back to your data source with a stored procedure	80
Saving changes using a QueryResolver	80

Coding stored procedures to handle data resolution	81
Saving changes with a ProcedureResolver	81
Example: Using InterBase stored procedures with return parameters	83
Resolving data from multiple tables	83
Considerations for the type of linkage between tables in the query	84
Table and column references (aliases) in a query string	84
Controlling the setting of the column properties	85
What if a table is not updatable?	85
How can the user specify that a table should never be updated?	85
Using DataSets with RMI (streamable data sets)	85
Example: Using streamable data sets	85
Using streamable DataSet methods	86
Customizing the default resolver logic	87
Understanding default resolving	87
Adding a QueryResolver component	87
Intercepting resolver events	88
Using resolver events	89
Writing a custom data resolver	90
Handling resolver errors	90
Resolving master-detail relationships	91

Chapter 9

Establishing a master-detail relationship 93

Defining a master-detail relationship	94
Creating an application with a master-detail relationship	94
Fetching details	97
Fetching all details at once	97
Fetching selected detail records on demand	97
Editing data in master-detail data sets	98
Steps to creating a master-detail relationship	99
Saving changes in a master-detail relationship	100
Resolving master-detail data sets to a JDBC data source	100

Chapter 10

Using data modules to simplify data access 103

Creating a data module using the design tools	104
Create the data module with the wizard	104
Add data components to the data module	104
Adding business logic to the data module	106
Using a data module	106
Adding a required library to a project	106
Referencing a data module in your application	107
Understanding the Use DataModule wizard	108
Creating data modules using the Data Modeler	109
Creating queries with the Data Modeler	109
Opening a URL	110
Beginning a query	110

Adding a Group By clause	112	Data type dependent patterns	150
Selecting rows with unique column values	113	Patterns for numeric data	150
Adding a Where clause	113	Patterns for date and time data	150
Adding an Order By clause	114	Patterns for string data	151
Editing the query directly	114	Patterns for boolean data	152
Testing your query	114	Presenting an alternate view of the data	152
Building multiple queries	115	Ensuring data persistence	153
Specifying a master-detail relationship	115	Making columns persistent	154
Saving your queries	116	Using variant data types	155
Generating database applications	117	Storing Java objects	155
Using a generated data module in your code	118		
Chapter 11		Chapter 13	
Filtering, sorting, and locating data	121	Using other controls and events	157
Retrieving data for the examples	122	Synchronizing visual components	157
Filtering data	124	Accessing data and model information from a UI component	158
Adding and removing filters	124	Displaying status information	158
Sorting data	127	Building an application with a JdbStatusLabel component	158
Sorting data in a JdbTable	127	Running the JdbStatusLabel application	159
Sorting data using the JBuilder visual design tools	128	Handling errors and exceptions	159
Understanding sorting and indexing	129	Overriding default DataSetException handling on controls	160
Sorting data in code	130		
Locating data	130	Chapter 14	
Locating data with a JdbNavField	130	Creating a distributed database application using DataSetData	161
Locating data programmatically	132	Understanding the sample distributed database application (using Java RMI and DataSetData)	161
Locating data using a DataRow	133	Setting up the sample application	162
Working with locate options	133	What is going on?	163
Locates that handle any data type	134	Passing metadata by DataSetData	163
Column order in the DataRow and DataSet	134	Deploying the application on multiple tiers	164
Chapter 12		Chapter 15	
Adding functionality to database applications	135	Database administration tasks	165
Using pick lists and lookups	136	Exploring database tables and metadata using the Database Pilot	165
Data entry with a pick list	136	Browsing database schema objects	166
Adding a pick list field	136	Setting up drivers to access remote and local databases	166
Removing a pick list field	137	Executing SQL statements	167
Create a lookup using a calculated column	138	Using the Explorer to view and edit table data	168
Create a lookup using the PickListDescriptor parameters	140	Using the Database Pilot for database administration tasks	170
Using calculated columns	142	Creating the SQL data source	170
Create a calculated column in the designer	143	Populating a SQL table with data using JBuilder	172
Aggregating data with calculated fields	144	Deleting tables in JBuilder	172
Example: Aggregating data with calculated fields	144	Monitoring database connections	172
Setting properties in the AggDescriptor	147	About the JDBC Monitor	172
Creating a custom aggregation event handler	147	Using the JDBC Monitor in a running application	174
Adding an Edit or Display Pattern for data formatting	148	Adding the MonitorButton to the Palette	174
Display masks	149	Using the MonitorButton Class from code	174
Edit masks	149	Understanding MonitorButton properties	174
Using masks for importing and exporting data	149		

Chapter 16

Tutorial: Importing and exporting data from a text file 175

Step 1: Creating the project	176
Step 2: Creating the text file	176
Step 3: Generating an application	177
Step 4: Adding DataExpress components to your application	177
Step 5: Adding dbSwing components to create a user interface	178
Step 6: Adding a JButton Swing component	180
Step 7: Compiling and running your application	181
Step 8: Using patterns for exporting numeric, date/time, and text fields	182

Chapter 17

Tutorial: Creating a basic database application 185

Step 1: Creating the project	187
Step 2: Generating an application	187
Step 3: Adding DataExpress components to your application	188

Step 4: Designing the columns for the application	190
Adding columns and editing column properties.	190
Specifying calculations for the calculated columns.	191
Step 5: Adding dbSwing components to create a user interface	192
Step 6: Aggregating data with calculated fields	194

Index 197

Figures

2.1	Diagram of a typical database application . . .	12	11.4	Sort property editor	128
2.2	JDataStore Explorer	20	11.5	Sorted application at runtime	129
4.1	Database component displayed in structure pane.	29	11.6	Sample application with JdbNavField . . .	131
4.2	Connection Descriptor dialog box.	30	12.1	Lookup application.	140
5.1	Query property editor	47	12.2	Calculated columns	144
5.2	Parameters page	49	12.3	Column designer.	154
5.3	Resource Bundle dialog	49	15.1	Database Pilot	165
8.1	UI for saving changes from a QueryDataSet.	79	15.2	Enter SQL page of the Database Pilot. . .	168
10.1	Data Modeler	109	15.3	JDBC Monitor	173
10.4	Group By page	112	15.4	JDBC Monitor with output	173
10.5	Where page.	113	16.1	Import/export database application	175
10.6	Order By page	114	16.2	Import/Export application at runtime. . . .	180
10.7	Link Queries dialog box.	115	16.3	Exporting data to text file application at runtime	181
10.8	Arrow showing relationship between queries	116	17.1	Basic database application	186
10.9	Editor showing code generated by Data Modeler	116	17.2	Query dialog box.	189
10.10	Data Module Application wizard	117	17.3	queryDataSet1 node expanded	189
11.1	Running database application	123	17.4	queryDataSet1 columns in the column designer	191
11.2	Application running filters	126	17.5	JdbTable component in the UI designer . .	193
11.3	Click on column header to sort at runtime .	127	17.6	Basic database application with navigation bar and status label	194
			17.7	Agg dialog box.	195

Tutorials

Importing and exporting data from a text file	175	Creating a basic database application	185
---	-----	---	-----

Introduction

Developing Database Applications provides information on using JBuilder's DataExpress database functionality to develop database applications. It also describes how to use dbSwing components to create a user interface (UI) for your application. Basic features that are commonly included in a database application are explained by example so you can learn by doing. Conceptual information is provided, followed with examples as applicable, with cross-references to more detailed information wherever possible.

Be sure to check for documentation additions and updates at <http://www.borland.com/techpubs/jbuilder>. Also, check the JBuilder online help. The information in the online help is more up-to-date than the printed material.

If you have questions about creating database applications using JBuilder, visit the database newsgroup at <news://newsgroups.borland.com/borland.public.jbuilder.database>. This newsgroup is dedicated to issues about writing database applications in JBuilder and is actively monitored by our support engineers as well as the JBuilder Development team. For discussions about dbSwing components, [borland.public.jbuilder.dbswing](news://newsgroups.borland.com/borland.public.jbuilder.dbswing) newsgroup is a good source for getting help creating database application UIs. A helpful DataExpress FAQ is currently located on the Borland Community Web site from <http://community.borland.com/>.

Note All versions of JBuilder provide direct access to SQL data through Sun's JDBC API. JBuilder Enterprise provides additional DataExpress components that greatly simplify development of database applications, as described in this book. Many of these components can be accessed easily from the DataExpress page of the component palette.

DataExpress stores data in memory. Most of the sample applications and tutorials described in this book use sample data that is stored in a JDataStore and is accessed through a JDBC driver. JDataStore's plug-in replacement for in-memory storage provides a permanent storage of data. JDataStore can be treated like any SQL database—you can connect to it as you would to any server, run SQL queries against it, etc. For more information on JDataStore, see the *JDataStore Developer's Guide*.

For an explanation of documentation conventions, see [“Documentation conventions” on page 6](#).

If you are unfamiliar with JBuilder, we suggest you start with *Getting Started with JBuilder*. If you are unfamiliar with Java, we suggest you start with *Getting Started with Java*.

Chapter summaries

This book details how database technologies and tools are surfaced in JBuilder and how you work with them in the IDE and the editor. It also explains how these technologies fit together in a database application. Choose one of the following topics for more information:

- [Chapter 2, “Understanding JBuilder database applications”](#)

Introduces the technologies, components, and tools used to create database applications in JBuilder, including elements of the DataExpress Component Library, Database Pilot, JDBC Monitor, Data sources, JDataStore, and InterBase.

- [Chapter 3, “Importing and exporting data from a text file”](#)

Explains how to provide data to your application from a text file, and how to save the data back to a text file or to a SQL data source.

- [Chapter 4, “Connecting to a database”](#)

Describes how to connect your database components to a server. Includes information on using JDBC and ODBC database drivers, and specific information for connecting to JDataStore and InterBase databases.

- [Chapter 5, “Retrieving data from a data source”](#)

Describes how to create a local copy of the data from your data source, and which DataExpress package components to use. This phase (called *providing*) makes the data available to your application.

- [Chapter 6, “Using stored procedures”](#)

Describes how to create and use stored procedures to execute SQL statements for providing or resolving data.

- [Chapter 7, “Working with columns”](#)

Describes how to make columns persistent, how to control the appearance and editing of column data, how to obtain metadata information, how to add a column to a data set, and how to define the order of display of columns.

- [Chapter 8, “Saving changes back to your data source”](#)

Describes how to save the data updates made by your JBuilder application back to the data source (a process called resolving). Covers multiple methods for resolving, including the basic resolver handling provided by DataExpress components, saving changes with stored procedures, resolving data from multiple tables, using `DataSet` objects with RMI, and customizing the default resolver logic.

- [Chapter 9, “Establishing a master-detail relationship”](#)

Provides information on linking two or more data sets to create a parent/child (or master-detail) relationship.

- [Chapter 10, “Using data modules to simplify data access”](#)

Describes how to use data modules to simplify data access in your applications, while at the same time standardizing database logic and business rules for all developers accessing the data. Also provides information on using the Data Modeler wizard to create data modules.

- [Chapter 11, “Filtering, sorting, and locating data”](#)

Provides information on how to implement data filtering, sorting, and locating in database applications, using standard DataExpress components and the JBuilder design tools. These features Explain the differences between these features, and provides a tutorial for each as well.

- [Chapter 12, “Adding functionality to database applications”](#)

Includes information on the following tasks:

- Formatting and parsing data with edit or display patterns
- Creating calculated columns
- Aggregating data (minimum, maximum, sum, count)
- Creating a lookup field
- Creating an alternate view of the data
- Creating persistent, or pre-defined, fields

- [Chapter 13, “Using other controls and events”](#)

Discusses additional methods for easing the development of the user-interface portion of your application. Includes information on displaying status information in your application, and application error handling.

- [Chapter 14, “Creating a distributed database application using DataSetData”](#)

Discusses using DataExpress components in a distributed object computing environment (using Java RMI).

- [Chapter 15, “Database administration tasks”](#)

Provides information about common database tasks, including:

- Browsing and editing data, tables, and database schema using the Database Pilot
- Creating and deleting tables
- Populating tables with data
- Monitoring JDBC traffic using the JDBC Monitor

Database tutorials

The following tutorials illustrate useful database application development techniques.

- [Chapter 16, “Tutorial: Importing and exporting data from a text file”](#)

Shows how to use the `TableDataSet` component to import and export data from a text file. This tutorial also shows how to use `dbSwing` components and the JBuilder design tools to build a user interface for the database application.

- [Chapter 17, “Tutorial: Creating a basic database application”](#)

Shows how to build a simple database application that connects to a SQL database. You will see how to set database connection properties, add a search field for locating data, and add calculated fields to total values in a column.

Database samples

There are many samples that demonstrate specific database application technologies or techniques. Most database-specific samples can be found in the following directories:

- `<jbuilder>/samples/DataExpress`: contains a variety of projects that demonstrate useful techniques for using DataExpress components to develop database applications.
- `<jbuilder>/samples/dbSwing`: contains projects that illustrate how to use `dbSwing` components to create effective user interfaces for database applications.

- `<jbuilder>/samples/JDataStore`: contains sample code, database files, and JBuilder projects to demonstrate the use of JDataStore databases and JDataStore database drivers with JBuilder. These sample files complement tutorials and samples discussed in the *JDataStore Developer's Guide*.

Many of the applications access data from the JDataStore sample database, `employee.jds`, and from the InterBase sample database `employee.gdb`. For more information on JDataStore, see the *JDataStore Developer's Guide*. For more information on InterBase Server, refer to its online documentation.

Throughout this guide, individual samples are referenced if they demonstrate a specific concept introduced in the text.

Note If you want to examine the sample applications in the JBuilder designer, you should build the project for each sample before bringing it into the designer. To build a project, choose Project|Rebuild Project.

Related documentation

The following Borland documentation contains useful information for developing database applications:

- *DataExpress Component Library Reference* is the online API documentation for DataExpress packages used for data access. It includes the following individual component package references:
 - *DataExpress Reference*:
Contains API documentation for the packages that provide basic data access. The `com.borland.dx.dataset` package provides general routines for data connectivity, management, and manipulation. The `com.borland.dx.sql.dataset` package provides data connectivity functionality that is JDBC specific. The `com.borland.dx.text` package contains classes and interfaces that control alignment and formatting of objects and the formatting of data and values. This package also handles formatting and parsing exceptions and input validation.
 - *dbSwing Reference*:
Contains API documentation for the `com.borland.dbswing` package, which contains components that allow you to make Swing components capable of accessing database data through DataExpress `DataSets`.
 - *JDataStore Reference*:
Contains API documentation for the packages used for connecting to and performing transactions with JDataStore databases. The `com.borland.datastore` package provides basic connectivity and transaction support for local JDataStore databases. The `com.borland.datastore.jdbc` package contains the JDBC interface for the DataStore, including the JDBC driver itself, and classes for implementing your own DataStore server for multi-user connections to the same DataStore. The `com.borland.datastore.javax.sql` package provides functionality for distributed transaction (XA) support. The classes in this package are used internally by other Borland classes. You should never use the classes in this package directly.
 - *Javax Classes Reference*:
Contains API documentation for the `com.borland.javax.sql` package, which provides implementations of JDBC 2.0 `DataSource` and connection pooling components. These classes can be used with any JDBC driver, but have additional functionality that is specific to the JDataStore JDBC driver.

- *InternetBeans Express Reference:*
Contains API documentation for the `com.borland.internetbeans` and `com.borland.internetbeans.taglib` packages that provide components and a JSP tag library for generating and responding to the presentation layer of a web application.
- *SQL Adapter Classes Reference:*
Contains API documentation for the `com.borland.sql` package. This package contains the `SQLAdapter` interface, which can be implemented by any JDBC class which can be adapted for improved performance.
- *SQL Tools Classes Reference:*
Contains API documentation for the `com.borland.sqltools` package, which contains classes for retrieving report output using SQL queries specified in XML format.
- *CORBA Express Reference:*
Contains API documentation for the `com.borland.cx` package, which contains CORBA connection classes for CORBA-based distributed applications.
- *DataExpress EJB Reference:*
Contains API documentation for the `com.borland.dx.ejb` package. This package contains DataExpress for EJB components that allow you to use entity beans with DataExpress DataSets to resolve and provide data. Some of these components can be added from the EJB page of the component palette in the UI designer.
- *XML Database Components Reference:*
Contains API documentation for XML database components in the `com.borland.jbuilder.xml.database.xmldbms`, `com.borland.jbuilder.xml.database.template`, and `com.borland.jbuilder.xml.database.common` packages. Many of the components in these packages can be added from the XML page of the component palette in the UI designer.
- *Developing Web Applications* contains information on using InternetBeans Express components to create web applications for data access. *Developing Web Applications* includes tutorials that show how to use InternetBeans Express components with JSPs and servlets.
- *Working with XML* explains how to use the XML model and template bean components for database queries and transfer of data between XML documents and databases. *Working with XML* also includes tutorials that demonstrate the use of the XML database components.
- *Developing Applications with Enterprise JavaBeans* describes how to use DataExpress for EJB components transfer data from entity beans deployed on a server to a client application and back again.
- “MIDP database programming” in *Developing Mobile Applications for MIDP* gives a brief overview of use of the Record Management System (RMS) for creating mobile database applications, and provides links to related information.
- *JDataStore Developer's Guide* contains comprehensive reference information to help you use JDataStore with database applications that you develop.

Documentation conventions

The Borland documentation for JBuilder uses the typefaces and symbols described in the following table to indicate special text.

Table 1.1 Typeface and symbol conventions

Typeface	Meaning
Bold	Bold is used for java tools, bmj (Borland Make for Java), bcj (Borland Compiler for Java), and compiler options. For example: javac , bmj , -classpath .
<i>Italics</i>	Italicized words are used for new terms being defined, for book titles, and occasionally for emphasis.
<i>Keycaps</i>	This typeface indicates a key on your keyboard, such as "Press <i>Esc</i> to exit a menu."
Monospaced type	Monospaced type represents the following: <ul style="list-style-type: none"> ■ text as it appears onscreen ■ anything you must type, such as "Type <code>Hello World</code> in the Title field of the Application wizard." ■ file names ■ path names ■ directory and folder names ■ commands, such as <code>SET PATH</code> ■ Java code ■ Java data types, such as <code>boolean</code>, <code>int</code>, and <code>long</code>. ■ Java identifiers, such as names of variables, classes, package names, interfaces, components, properties, methods, and events ■ argument names ■ field names ■ Java keywords, such as <code>void</code> and <code>static</code>
[]	Square brackets in text or syntax listings enclose optional items. Do not type the brackets.
< >	<p>Angle brackets are used to indicate variables in directory paths, command options, and code samples.</p> <p>For example, <code><filename></code> may be used to indicate where you need to supply a file name (including file extension), and <code><username></code> typically indicates that you must provide your user name.</p> <p>When replacing variables in directory paths, command options, and code samples, replace the entire variable, including the angle brackets (<code>< ></code>). For example, you would replace <code><filename></code> with the name of a file, such as <code>employee.jds</code>, and omit the angle brackets.</p> <p>Note: Angle brackets are used in HTML, XML, JSP, and other tag-based files to demarcate document elements, such as <code></code> and <code><ejb-jar></code>. The following convention describes how variable strings are specified within code samples that are already using angle brackets for delimiters.</p>
<i>Italics, serif</i>	This formatting is used to indicate variable strings within code samples that are already using angle brackets as delimiters. For example, <code><url="jdbc:borland:jbuilder\\samples\\guestbook.jds"></code>
...	In code examples, an ellipsis (...) indicates code that has been omitted from the example to save space and improve clarity. On a button, an ellipsis indicates that the button links to a selection dialog box.

JBuilder is available on multiple platforms. See the following table for a description of platform conventions used in the documentation.

Table 1.2 Platform conventions

Item	Meaning
Paths	Directory paths in the documentation are indicated with a forward slash (/). For Windows platforms, use a backslash (\).
Home directory	The location of the standard home directory varies by platform and is indicated with a variable, <home>. <ul style="list-style-type: none"> ■ For UNIX and Linux, the home directory can vary. For example, it could be /user/<username> or /home/<username> ■ For Windows NT, the home directory is C:\Winnt\Profiles\<username> ■ For Windows 2000 and XP, the home directory is C:\Documents and Settings\<username>
Screen shots	Screen shots reflect the Borland Look & Feel on various platforms.

Developer support and resources

Borland provides a variety of support options and information resources to help developers get the most out of their Borland products. These options include a range of Borland Technical Support programs, as well as free services on the Internet, where you can search our extensive information base and connect with other users of Borland products.

Contacting Borland Developer Support

Borland offers several support programs for customers and prospective customers. You can choose from several categories of support, ranging from free support upon installation of the Borland product, to fee-based consultant-level support and extensive assistance.

For more information about Borland's developer support services, see our web site at <http://www.borland.com/devsupport/>, call Borland Assist at (800) 523-7070, or contact our Sales Department at (831) 431-1064.

When contacting support, be prepared to provide complete information about your environment, the version of the product you are using, and a detailed description of the problem.

For support on third-party tools or documentation, contact the vendor of the tool.

Online resources

You can get information from any of these online sources:

World Wide Web	http://www.borland.com/ http://info.borland.com/techpubs/jbuilder/
Electronic newsletters	To subscribe to electronic newsletters, use the online form at: http://www.borland.com/products/newsletters/index.html

World Wide Web

Check the JBuilder page of the Borland website, www.borland.com/jbuilder, regularly. This is where the Java Products Development Team posts white papers, competitive analyses, answers to frequently asked questions, sample applications, updated software, updated documentation, and information about new and existing products.

You may want to check these URLs in particular:

- <http://www.borland.com/jbuilder/> (updated software and other files)
- <http://info.borland.com/techpubs/jbuilder/> (updated documentation and other files)
- <http://bdn.borland.com/> (contains our web-based news magazine for developers)

Borland newsgroups

When you register JBuilder you can participate in many threaded discussion groups devoted to JBuilder. The Borland newsgroups provide a means for the global community of Borland customers to exchange tips and techniques about Borland products and related tools and technologies.

You can find user-supported newsgroups for JBuilder and other Borland products at <http://www.borland.com/newsgroups/>.

Usenet newsgroups

The following Usenet groups are devoted to Java and related programming issues:

- news:comp.lang.java.advocacy
- news:comp.lang.java.announce
- news:comp.lang.java.beans
- news:comp.lang.java.databases
- news:comp.lang.java.gui
- news:comp.lang.java.help
- news:comp.lang.java.machine
- news:comp.lang.java.programmer
- news:comp.lang.java.security
- news:comp.lang.java.softwaretools

Note These newsgroups are maintained by users and are not official Borland sites.

Reporting bugs

If you find what you think may be a bug in the software, please report it to Borland at one of the following sites:

- Support Programs page at <http://www.borland.com/devsupport/namerica/>. Click the Information link under “Reporting Defects” to open the Welcome page of Quality Central, Borland’s bug-tracking tool.
- Quality Central at <http://qc.borland.com>. Follow the instructions on the Quality Central page in the “Bugs Report” section.
- Quality Central menu command on the main Tools menu of JBuilder (Tools|Quality Central). Follow the instructions to create your QC user account and report the bug. See the Borland Quality Central documentation for more information.

When you report a bug, please include all the steps needed to reproduce the bug, including any special environmental settings you used and other programs you were

using with JBuilder. Please be specific about the expected behavior versus what actually happened.

If you have comments (compliments, suggestions, or issues) for the JBuilder documentation team, you may email jpgpubs@borland.com. This is for documentation issues only. Please note that you must address support issues to developer support.

JBuilder is made by developers for developers. We really value your input.

Understanding JBuilder database applications

A database application is any application that accesses stored data and allows you to view and perhaps modify or manipulate that data. In most cases, the data is stored in a database. However, data can also be stored in files as text, or in some other format. JBuilder allows you to access this information and manipulate it using properties, methods, and events defined in the `DataSet` packages of the `DataExpress` Component Library in conjunction with the `dbSwing` package.

A database application that requests information from a data source such as a database is known as a client application. A DBMS (Database Management System) that handles data requests from various clients is known as a database server.

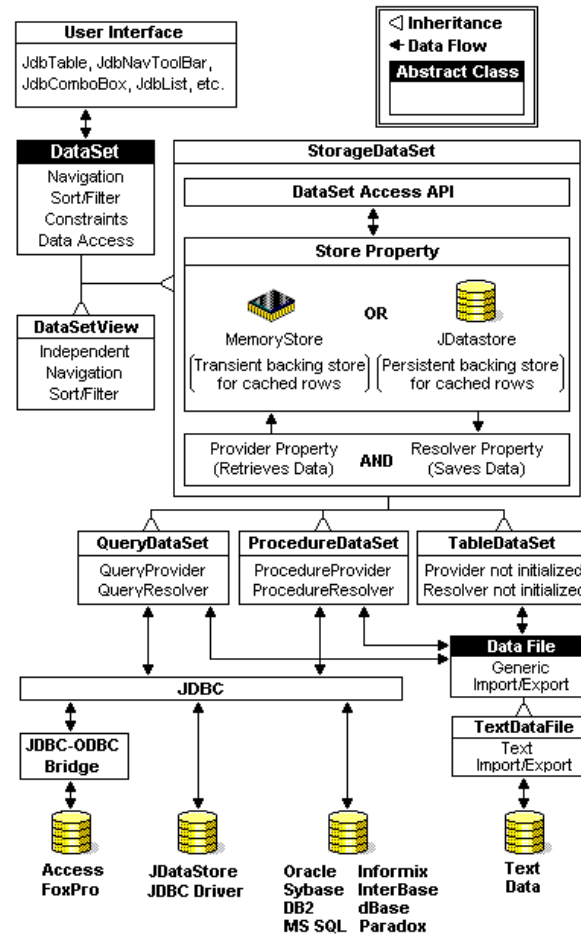
Database application architecture

JBuilder's `DataExpress` architecture is focused on building all-Java client-server applications, applets, servlets, and JavaServer Pages (JSP) for the inter- or intranet. Because applications you build in JBuilder are all-Java at run time, they are cross-platform.

JBuilder applications communicate with database servers through the JDBC API, the Sun database connectivity specification. JDBC is the all-Java industry standard API for accessing and manipulating database data. JBuilder database applications can connect to any database that has a JDBC driver.

The following diagram illustrates a typical database application and the layers from the client JBuilder *DataExpress* database application to the data source:

Figure 2.1 Diagram of a typical database application



The next section, “[DataExpress components](#),” discusses the components of the *DataExpress* architecture in more detail.

DataExpress components

DataExpress is a package, `com.borland.dx.dataset`, of Borland classes and interfaces that provide basic data access. This package also defines base provider and resolver classes as well as an abstract *DataSet* class that is extended to other *DataSet* objects. These classes provide access to information stored in databases and other data sources. This package includes functionality covering the three main phases of data handling:

- **Providing**

General functionality to obtain data and manage local data sets. (JDBC specific connections to remote servers are handled by classes in the `com.borland.dx.sql.dataset` package.)

- **Manipulation**

Navigation and editing of the data locally.

- **Resolving**

General routines for the updating of data from the local *DataSet* back to the original source of the data. (Resolving data changes to remote servers through JDBC is handled by classes in the `com.borland.dx.sql.dataset` package.)

Key features and benefits

DataExpress components were designed to be modular to allow the separation of key functionality. This design allows the DataExpress components to handle a broad variety of applications. Modular aspects of the DataExpress architecture include:

- Core DataSet functionality

This is a collection of data handling functionality available to applications using DataExpress. Much of this functionality can be applied using declarative property and event settings. Functionality includes navigation, data access/update, ordering/filtering of data, master-detail support, lookups, constraints, defaults, etc.

- Data source independence

The retrieval and update of data from a data source, such as an Oracle or Sybase server, is isolated to two key interfaces: Provider/Resolver. By cleanly isolating the retrieval and updating of data, it is easy to create new Provider/Resolver components for new data sources. There are two Provider/Resolver implementations for standard JDBC drivers that provide access to databases such as Oracle, Sybase, Informix, InterBase, DB2, MS SQL Server, Paradox, dBase, FoxPro, Access, and other databases. You can also create custom Provider/Resolver component implementations for EJB, application servers, SAP, BAAN, IMS, CICS, etc.

- Pluggable storage

When data is retrieved from a Provider it is cached inside the DataSet. All edits made to the cached DataSet are tracked so that Resolver implementations know what needs to be updated back to the data source. DataExpress provides two options for this caching storage: MemoryStore (the default), and JDataStore.

MemoryStore caches all data and data edits in memory. JDataStore uses an all Java, small footprint, high performance, embeddable database to cache data and data edits. The JDataStore is ideally suited for disconnected/mobile computing, asynchronous data replication, and small footprint database applications.

- Data binding support for visual components

DataExpress DataSet components provide a powerful programmatic interface, as well as support for direct data binding to data-aware components by way of point and click property settings made in a visual designer. JBuilder ships with Swing-based visual components that bind directly to DataSet components.

The benefits of using the modular DataExpress Architecture include:

- Network computing

As mentioned, the Provider/Resolver approach isolates interactions with arbitrary data sources to two clean points. There are two other benefits to this approach:

- The Provider/Resolver can be easily partitioned to a middle tier. Since Provider/Resolver logic typically has a transactional nature, it is ideal for partitioning to a middle tier.
- It is a “stateless” computing model that is ideally suited to network computing. The connection between the DataSet component client and the data source can be disconnected after providing. When changes need to be saved back to the data source, the connection need only be re-established for the duration of the resolving transaction.

- Rapid development of user interfaces

Since DataSet components can be bound to data-aware components with a simple property setting, they are ideally suited for rapidly building database application user interfaces.

- Mobile computing

With the introduction of the `DataStore` component, DataExpress applications have a persistent, portable database. The `DataStore` can contain multiple `DataSet` components, arbitrary files, and Java Objects. This allows a complete application state to be persisted in a single file. `DataSet` components have built-in data replication technology for saving and reconciling edits made to replicated data back to a data source.

- Embedded applications

The small footprint, high performance `JDataStore` database is ideal for embedded applications and supports the full functionality and semantics of the `DataSet` component.

For more information on the DataExpress architecture, visit the Borland web site at <http://www.borland.com/jbuilder/> for a white paper on this topic.

Overview of the DataExpress components

The core functionality required for data connectivity is contained in the `com.borland.dx.dataset`, `com.borland.dx.sql.dataset`, and `com.borland.datastore` packages. The components in these packages encapsulate both the connection between the application and its source of the data, as well as the behavior needed to manipulate the data. The features provided by these packages include that of database connectivity as well as data set functionality.

The main classes and components in the Borland database-related packages are listed in the table below, along with a brief description of the component or class. The right-most column of this table lists frequently used properties of the class or component. Some properties are themselves objects that group multiple properties. These complex property objects end with the word `Descriptor` and contain key properties that (typically) must be set for the component to be usable.

Component/Class	Description	Frequently used properties
Database	<p>A required component when accessing data stored on a remote server, the <code>Database</code> component manages the JDBC connection to the SQL server database.</p> <p>See Chapter 4, “Connecting to a database” for more description and instructions for using this component.</p>	<p>The <code>ConnectionDescriptor</code> object stores connection properties of driver, URL, user name, and password. Accessed using the <code>connection</code> property.</p>
DataSet	<p>An abstract class that provides basic data set behavior, <code>DataSet</code> also provides the infrastructure for data storage by maintaining a two-dimensional array that is organized by rows and columns. It has the concept of a current row position, which allows you to navigate through the rows of data and manages a “pseudo record” that holds the current new or edited record until it is posted into the <code>DataSet</code>. Because it extends <code>ReadWriteRow</code>, <code>DataSet</code> has methods to get and put field values.</p>	<p>The <code>SortDescriptor</code> object contains properties that affect the order in which data is accessed and displayed in a UI component. Set using the <code>sort</code> property. See “Sorting data” on page 127 for usage information.</p> <p>The <code>MasterLinkDescriptor</code> object contains properties for managing master-detail relationships between two <code>DataSet</code> components. Accessed using the <code>masterLink</code> property on the detail <code>DataSet</code>. See Chapter 9, “Establishing a master-detail relationship” for usage instructions.</p>

Component/Class	Description	Frequently used properties
StorageDataSet	<p>A class that extends <code>DataSet</code> by providing implementation for storage of the data and manipulation of the structure of the <code>DataSet</code>.</p> <p>You fill a <code>StorageDataSet</code> component with data by extracting information from a remote database (such as InterBase or Oracle), or by importing data stored in a text file. This is done by instantiating one of its subclasses: <code>QueryDataSet</code>, <code>ProcedureDataSet</code>, or <code>TableDataSet</code>.</p>	<p>The <code>tableName</code> property specifies the data source of the <code>StorageDataSet</code> component.</p> <p>The <code>maxRows</code> property defines the maximum number of rows that the <code>DataSet</code> can initially contain.</p> <p>The <code>readOnly</code> property controls write-access to the data.</p>
DataStore	<p>The <code>DataStore</code> component provides a replacement for <code>MemoryStore</code> that gives a permanent storage of data. A <code>JDataStore</code> provides high performance data caching and compact persistence for DataExpress <code>DataSets</code>, arbitrary files, and Java Objects. The <code>DataStore</code> component uses a single file to store one or more data streams. A <code>JDataStore</code> file has a hierarchical directory structure that associates a name and directory status with a particular data stream. <code>JDataStore</code> can be treated like any SQL database—you can connect to it as you would to any server, run SQL queries against it, etc.</p> <p>See “JDataStore and JBuilder” on page 19, and the <i>JDataStore Developer's Guide</i>, for more description of the <code>DataStore</code> component.</p>	<p>Caching and persisting <code>StorageDataSet</code> components in a <code>DataStore</code> is accomplished through two required property settings on a <code>StorageDataSet</code> called <code>store</code> and <code>storeName</code>. By default, all <code>StorageDataSets</code> use a <code>MemoryStore</code> if the <code>store</code> property is not set. Currently <code>MemoryStore</code> and <code>DataStore</code> are the only implementations for the <code>store</code> property. The <code>storeName</code> property is the unique name associated with this <code>StorageDataSet</code> in the <code>DataStore</code>.</p>
DataStoreDriver	<p><code>DataStoreDriver</code> is the JDBC driver for the <code>JDataStore</code>. The driver supports both local and remote access. Both types of access require a user name (any string, with no setup required). A non-null password is required by default.</p>	
QueryDataSet	<p>The <code>QueryDataSet</code> component stores the results of a query string executed against a server database. This component works with the <code>Database</code> component to connect to SQL server databases, and runs the specified query with parameters (if any). Once the resulting data is stored in the <code>QueryDataSet</code> component, you can manipulate the data using the <code>DataSet</code> API.</p> <p>See “Querying a database” on page 46 for more description and instructions for using this component.</p>	<p>The <code>QueryDescriptor</code> object contains the SQL query statement, query parameters, and database connection information. Accessed using the <code>query</code> property.</p>
ProcedureDataSet	<p>The <code>ProcedureDataSet</code> component holds the results of a stored procedure executed against a server database. This component works with the <code>Database</code> component in a manner similar to the <code>QueryDataSet</code> component.</p> <p>See Chapter 6, “Using stored procedures” for more information and usage instructions for this component.</p>	<p>The <code>ProcedureDescriptor</code> object contains the SQL statement, parameters, database component, and other properties. Accessed using the <code>procedure</code> property of the <code>ProcedureDataSet</code> component.</p>
TableDataSet	<p>Use this component when importing data from a text file. This component extends the <code>DataSet</code> class. It mimics SQL server functionality, without requiring a SQL server connection.</p> <p>See Chapter 3, “Importing and exporting data from a text file” and the related tutorial, Chapter 16, “Tutorial: Importing and exporting data from a text file” for more description and usage instructions for this component.</p>	<p>The (inherited) <code>dataFile</code> property specifies the file name from which to load data into the <code>DataSet</code> and to save the data to.</p>
DataSetView	<p>This component presents an alternate “view” of the data in an existing <code>StorageDataSet</code>. It has its own (inherited) <code>sort</code> property, which, if given a new value, allows a different ordered presentation of the data. It also has filtering and navigation capabilities that are independent of its associated <code>StorageDataSet</code>.</p> <p>See “Presenting an alternate view of the data” on page 152 for more description and usage instructions for this component.</p>	<p>The <code>storageDataSet</code> property indicates the component which contains the data of which the <code>DataSetView</code> presents a view.</p>

Component/Class	Description	Frequently used properties
Column	<p>A <code>Column</code> represents the collection from all rows of a particular item of data, for example, all the <code>Name</code> values in a table. A <code>Column</code> gets its value when a <code>DataSet</code> is instantiated or as the result of a calculation.</p> <p>The <code>Column</code> is managed by its <code>StorageDataSet</code> component.</p> <p>See Chapter 7, “Working with columns” for more description and usage instructions for this component.</p>	<p>You can conveniently set properties at the <code>Column</code> level so that settings which affect the entire column of data can be set at one place, for example, <code>font</code>. JBuilder design tools include access to column-level properties by double-clicking any <code>StorageDataSet</code> in the content pane, then selecting the <code>Column</code> that you want to work with. The selected <code>Column</code> component's properties and events display in either the column designer (properties only) or in the Inspector and can be edited in either place.</p>
DataRow	<p>The <code>DataRow</code> component is a collection of all <code>Column</code> data for a single row where each row is a complete record of information. The <code>DataRow</code> component uses the same columns of the <code>DataSet</code> it was constructed with. The names of the columns in a <code>DataRow</code> are field names.</p> <p>A <code>DataRow</code> is convenient to work with when comparing the data in two rows or when locating data in a <code>DataSet</code>. It can be used in all <code>DataSet</code> methods that require a <code>ReadRow</code> or <code>ReadWriteRow</code>.</p>	
ParameterRow	<p>The <code>ParameterRow</code> component has a <code>Column</code> for each column of the associated data set that you may want to query. Place values you want the query to use in the <code>ParameterRow</code> and associate them with the query by their parameter names (which are the <code>ParameterRow</code> column names).</p> <p>See “Using parameterized queries to obtain data from your database” on page 53 for more description and instructions for using this component.</p>	
DataModule	<p>The <code>DataModule</code> is an interface in the <code>com.borland.dx.dataset</code> package. A class that implements <code>DataModule</code> will be recognized by the JBuilder designer as a class that contains various <code>dataset</code> components grouped into a data model. You create a new, empty data module by double-clicking the Data Module icon in the object gallery (File New). Then using the component palette and content pane, you place into it various <code>DataSet</code> objects, and provide connections, queries, sorts, and custom business rules logic. Data modules simplify reuse and multiple use of collections of <code>DataSet</code> components. For example, one or more UI classes in your application can use a shared instance of your custom <code>DataModule</code>.</p> <p>See Chapter 10, “Using data modules to simplify data access” for more description and instructions for using this component.</p>	

There are many other classes and components in the `com.borland.dx.dataset`, `com.borland.dx.sql.dataset`, and `com.borland.datastore` packages as well as several support classes in other packages such as the `util` and `view` packages. Detailed information on the packages and classes of DataExpress Library can be found in the *DataExpress Component Library Reference* documentation.

DataExpress for EJB components

**This is a feature of
JBuilder Enterprise**

The DataExpress for EJB package, `com.borland.dx.ejb` package contains the DataExpress for EJB components. These components allow you to provide data from EJB entity beans to DataExpress DataSets, and then resolve changes made to the DataSets back to entity beans.

The DataExpress for EJB package is not covered in this book. For more information about using components from the DataExpress for EJB package to develop data-aware enterprise applications, see “Using the DataExpress for EJB components” in the *Developing Applications with Enterprise JavaBeans*. For reference information, refer to the API documentation for the `com.borland.dx.ejb` package.

InternetBeans Express

**This is a feature of
JBuilder Developer
and Enterprise**

The InternetBeans Express package, `com.borland.internetbeans`, provides components and a JSP tag library for generating and responding to the presentation layer of a web application.

The InternetBeans Express package is not covered in this book. For more information about using components from InternetBeans Express package to develop data-aware JSP and servlet applications, see “InternetBeans Express” in the *Developing Web Applications*. For reference information, refer to the API documentation for the `com.borland.internetbeans` package.

XML database components

**This is a feature of
JBuilder Developer
and Enterprise**

JBuilder’s XML database components support the development of XML database applications. The components can be added to your application from the XML page of the component palette in the UI designer. There are model-based components and template-based components. Model-based components use a map document that determines how the data transfers between an XML structure and the database metadata. To use template-based components, you supply a SQL statement, and the component generates an XML document. The SQL you provide serves as the template that is replaced in the XML document as the result of applying the template.

The use of XML database components is not covered in this book. For more information, see “Using JBuilder’s XML database components” in *Working with XML*. For reference information, refer to the API documentation for the `com.borland.jbuilder.xml.database.common`, `com.borland.jbuilder.xml.database.template`, `com.borland.jbuilder.xml.database.xmldbms` packages.

dbSwing

The `dbSwing` package allows you to build database applications that take advantage of the Java Swing component architecture. In addition to pre-built, data-aware subclasses of most Swing components, `dbSwing` also includes several utility components designed specifically for use in developing DataExpress and JDataStore-based applications.

To create a database application, you first need to connect to a database and provide data to a `DataSet`. “Retrieving data for the examples” on page 122 sets up a query that can be used as a starting point for creating a database application and a basic user interface.

To use the data-aware dbSwing components,

- 1 Open the Frame file, and select the Design tab.
- 2 Select one of the dbSwing pages: dbSwing, More dbSwing, or dbSwing Models.
- 3 Click a component on the component palette, and click in the UI designer to place the component in the application.
- 4 Select the component in the component tree or the UI designer.

Depending on the type of component, and the `layout` property for the `contentPane` containing the component, the designer displays black sizing nibs on the edges of a selected component.

Some of the component's (`JdbNavToolBar` and `JdbStatusLabel`) automatically bind to whichever data set has focus. For others (like `JdbTable`), set the component's `dataSet` and/or `columnName` properties in the Inspector to bind the component to an instantiated `DataSet`.

The following list contains a few of the dbSwing components available from the dbSwing page of the component palette:

- `TableScrollPane`
- `JdbTable`
- `JdbNavToolBar`
- `JdbStatusLabel`
- `JdbTextArea`
- `JdbComboBox`
- `JdbLabel`
- `JdbList`
- `JdbTextPane`
- `JdbTextField`

With increased functionality and data-aware capabilities, dbSwing offers significant advantages over Swing. Also, dbSwing is entirely lightweight, provides look-and-feel support for multiple platforms, and has strong conformance to Swing standards. Using dbSwing components, you can be sure all your components are lightweight.

For more information about the dbSwing package, refer to the dbSwing API documentation.

Data modules and the Data Modeler

Data modules provide a container for data access components. Data modules simplify database application development by modularizing your code and separating the database access logic and business rules in your applications from the user interface logic. You can also maintain control over the use of the data module by delivering only the class files to application developers.

The Data Modeler is a wizard that can help you build data modules that encapsulate a connection to a database and the queries to be run against the database.

For more information about data modules and the Data Modeler, refer to [Chapter 10](#), “Using data modules to simplify data access.”

Database Pilot

**This is a feature of
JBuilder Developer
and Enterprise**

The Database Pilot (Tools|Database Pilot) is a hierarchical database browser that also allows you to edit data.

The Database Pilot presents JDBC-based meta-database information in a two-paned window. The left pane contains a tree that hierarchically displays a set of databases

and its associated tables, views, stored procedures, and metadata. The right pane is a multi-page display of descriptive information for each node of the tree. In certain cases, you can edit data in the right pane as well.

For more information about the Database Pilot, refer to [Chapter 15, “Database administration tasks.”](#)

JDBC Monitor

**This is a feature of
JBuilder Developer
and Enterprise**

The JDBC Monitor (Tools|JDBC Monitor) is a graphical tool that can be used to monitor JDBC traffic. JDBC Monitor will monitor any JDBC driver (that is, any subclass of `java.sql.Driver`) while it is in use by JBuilder. The JDBC Monitor monitors all output directly from the JDBC driver.

For more information about the JDBC Monitor, including usage instructions, see [“Monitoring database connections” on page 172.](#)

JDataStore and JBuilder

JDataStore is a high-performance, small-footprint, all Java multifaceted data storage solution. JDataStore is,

- An embedded relational database, with both JDBC and DataExpress interfaces, that supports non-blocking transactional multi-user access with crash recovery
- An object store, for storing serialized objects, datasets, and other file streams
- A JavaBean component, that can be manipulated with visual bean builder tools like JBuilder

An all-Java visual JDataStore Explorer helps you manage your datastores.

For the most complete and up-to-date information on using a `DataStore`, refer to the *JDataStore Developer's Guide*.

When to use JDataStore versus JDBC drivers

There are unique advantages to configuring a database application to access a relational database management system with JDBC drivers versus configuring the application to use JDataStore. The following sections outline some of the advantages associated with either approach.

You may want to use a JDBC driver to:

- Use a standards-based JDBC API
- Work with live SQL data—you can use a `QueryProvider` to query a SQL database and work with live data, saving changes as necessary
- Take advantage of remote access using `RemoteJDBC`

Note JDataStore can be used with or without JDBC drivers. In fact, most of the samples and tutorials referenced in this book, JDataStore is used with JDBC drivers. See the “Offline editing with JDataStore” tutorial in the *JDataStore Developer's Guide* for an example of how a `DataStore` component can be used for offline editing of data, instead of making a JDBC connection to a JDataStore server.

You may want to use a JDataStore to,

- Work off-line—you can save and edit data within the JDataStore file system, resolve edits back when you are reconnected to the data source
- Store objects, as well as data
- Work with larger sets of data

Additional advantages of a JDataStore

You may want to use a JDataStore for any of the following reasons:

- **Organization.**

To organize an application's `StorageDataSet` components, files, and serialized JavaBean/Object state into a single all Java, portable, compact, high-performance, persistent storage.

- **Asynchronous data replication.**

For mobile/offline computing models, `StorageDataSet` has support for resolving/reconciling edited data retrieved from an arbitrary data source (i.e. JDBC, Application Server, SAP, BAAN, etc.).

- **Embedded applications.**

JDataStore foot print is very small. `StorageDataSet` components also provide excellent data binding support for data-aware UI components.

- **Performance.**

To increase performance and save memory for a large `StorageDataSet`. `StorageDataSet` components using `MemoryStore` will have a small performance edge over `DataStore` for small number of rows. `DataStore` stores `StorageDataSet` data and indexes in an extremely compact format. As the number of rows in a `StorageDataSet` increases, the `StorageDataSet` using a `DataStore` provides better performance and requires much less memory than a `StorageDataSet` using a `MemoryStore`.

For more information on using JDataStores, refer to the *JDataStore Developer's Guide*.

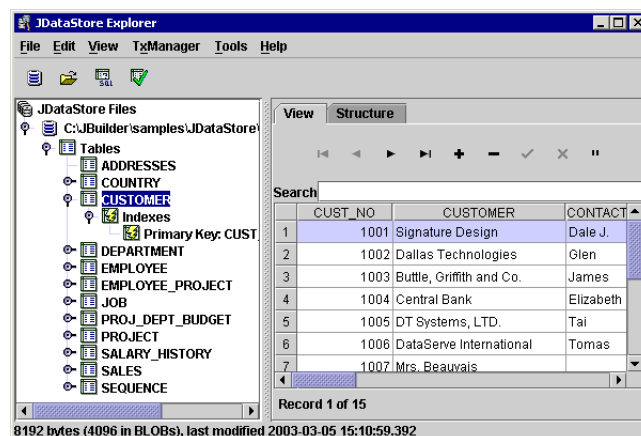
Using the JDataStore Explorer

Using the JDataStore Explorer, you can,

- Examine the contents of a `DataStore`. The store's directory is shown in a tree control, with each data set and its indexes grouped together. When a data stream is selected in the tree, its contents are displayed (assuming it's a file type like text file, .gif, or data set, for which the Explorer has a viewer).
- Perform many store operations without writing code. You can create a new JDataStore, import delimited text files into data sets, import files into file streams, delete indexes, delete data sets or other data streams, and verify the integrity of the JDataStore.
- Manage queries that provide data into data sets in the store, edit the data sets, and save changes back to server tables.

Use the Tools|JDataStore Explorer menu command to launch the JDataStore Explorer.

Figure 2.2 JDataStore Explorer



JDataStore explorer operations

To create a new JDataStore,

- 1 Open the JDataStore Explorer by choosing Tools|JDataStore Explorer.
- 2 Choose File|New or click the New JDataStore button.
- 3 Enter a name for the new store and choose OK. The store is created and opened in the Explorer.

To import a text file into a data set,

- 1 Choose Tools|Import|Text Into Table.
- 2 Supply the input text file and the store name of the data set to be created.

The contents of the text file must be in the delimited format that JBuilder exports to, and there must be a SCHEMA (.schema) file with the same name in the directory to define the structure of the target data set (to create a .schema file, see [“Exporting data” on page 25](#)). The default store name is the input file name, including the extension. Since this operation creates a data set, not a file stream, you’ll probably want to omit the extension from the store name.

- 3 Click OK.

To import a file into a file stream,

- 1 Choose Tools|Import|File.
- 2 Supply an input file name and the store name, and click OK.

To verify the open JDataStore, choose Tools|Verify JDataStore or click the Verify JDataStore button.

The entire store is verified and the results are displayed in the Verifier Log window. After you’ve closed the log window, you view it again by choosing View|Verifier Log.

For more information on using the JDataStore Explorer, refer to “JDataStore Administration” in the *JDataStore Developer’s Guide*.

InterBase and JBuilder

Borland InterBase is a high-performance, cross-platform, SQL standards-compliant relational database. InterBase includes its own version of the employee database, `employee.gdb`, so you can easily use InterBase instead of JDataStore in the samples and tutorials. For information on setting up Interbase and InterClient for use in the tutorials, see [“Connecting to a database using InterClient JDBC drivers” on page 35](#).

For more information about InterBase or to download a free trial version of InterBase, see the Borland web site at <http://www.borland.com/interbase/index.html>.

Importing and exporting data from a text file

In JBuilder, a `TableDataSet` component is used to store data imported from a text file. Once the data is provided to the data set, it can be viewed and modified. To save changes back to the text file, export the data back to the text file.

To import data from a text file, use the `TextDataFile` component to provide the location of the text file and parameters specific to its structure. Use a `StorageDataSet`, such as a `TableDataSet` component, to store the data locally for viewing and editing. Create `Column` objects so the `TableDataSet` knows the type of data and the name of the field for each column of data.

Columns of a `TableDataSet` are defined by adding columns in the Source window, the UI designer, or by loading a text file with a valid SCHEMA (`.schema`) file. This topic discusses the first two options. Importing data using an existing SCHEMA file is discussed in [Chapter 16, “Tutorial: Importing and exporting data from a text file.”](#) Your text file has a valid SCHEMA file only if it has previously been exported by JBuilder.

These are the topics covered:

- [“Adding columns to a TableDataSet in the editor” on page 23](#)
- [“Importing formatted data from a text file” on page 24](#)
- [“Retrieving data from a JDBC data source” on page 24](#)
- [“Exporting data” on page 25](#)

Adding columns to a TableDataSet in the editor

You can add columns to the `TableDataSet` in two ways: visually in the UI designer and with code in the editor on the Source tab. Adding columns in the UI designer is covered in [Chapter 16, “Tutorial: Importing and exporting data from a text file.”](#) If you previously exported to a text file, JBuilder created a SCHEMA file that provides column definitions when the text file is next opened; therefore, you do not need to add columns manually.

To add the columns using the editor, you define new `Column` objects in the class definition for `Frame1.java` as follows:

- 1 Select `Frame1.java` in the content pane, then select the Source tab. You will see the class definition in the Source window. Add the follow line of code:

```
Column column1 = new Column();
Column column2 = new Column();
```

- 2 Find the `jbInit()` method in the source code. Define the name of the column and the type of data that will be stored in the column, as follows:

```
column1.setColumnName("my_number");
column1.setDataType(com.borland.dx.dataset.Variant.SHORT);

column2.setColumnName("my_string");
column2.setDataType(com.borland.dx.dataset.Variant.STRING);
```

- 3 Add the new columns to the `TableDataSet` in the same source window and same `jbInit()` method, as follows:

```
tableDataSet1.setColumns(new Column[] { column1,column2 } );
```

- 4 Compile the application to bind the new `Column` objects to the data set, then add any visual components.

Importing formatted data from a text file

Data in a column of the text file may be formatted for exporting data in a way that prevents you from importing the data correctly. You can solve this problem by specifying a pattern to be used to read the data in an `exportDisplayMask`. The `exportDisplayMask` property is used for importing data when there is no SCHEMA file associated with the text file. If there is a SCHEMA file, its settings have precedence. The syntax of patterns is defined in “String-based patterns (masks)” in the *DataExpress Component Library Reference*.

Date and number columns have default display and edit patterns. If you do not set the properties, default edit patterns are used. The default patterns come from the `java.text.resources.LocaleElements` file that matches the column's default locale. If no locale is set for the column, the data set's locale is used. If no locale is set for the data set, the default system locale is used. The default display for a floating point number shows three decimal places. If you want more decimal places, you must specify a mask.

Retrieving data from a JDBC data source

The following code is an example of retrieving data from a JDBC data source into a `TextDataFile`. Once the data is in a `TextDataFile`, you can use a `StorageDataSet`, such as a `TableDataSet` component, to store the data locally for viewing and editing. For more information on how to do this, see [Chapter 16, “Tutorial: Importing and exporting data from a text file.”](#)

```
Database db = new Database();
db.setConnection(new
    com.borland.dx.sql.dataset.ConnectionDescriptor("jdbc:oracle:thin:@" +
        datasource, username, password));
QueryDataSet qds = new QueryDataSet();
qds.setQuery(new com.borland.dx.sql.dataset.QueryDescriptor(db, "SELECT
    * FROM THETABLE", null, true, Load.ALL));
TextDataFile tdf = new TextDataFile();
tdf.setFileName("THEDATA.TXT");
tdf.save(qds);
```

This code produces a data file and an associated SCHEMA file.

You can use this type of data access to create a database table backup-and-restore application that works from the command line, for example. To save this information back to the JDBC data source, see [“Saving changes loaded from a TextDataFile to a JDBC data source” on page 26](#).

Exporting data

Exporting data, or *saving data to a text file*, saves all of the data in the current view to the text file, overwriting the existing data. This topic discusses several ways to export data. You can export data that has been imported from a text file back to that file or to another file. You can export data from a `QueryDataSet` or a `ProcedureDataSet` to a text file. Or you can resolve data from a `TableDataSet` to an existing SQL table.

Exporting data to a text file is handled differently than resolving data to a SQL table. Both `QueryDataSet` and `TableDataSet` are `StorageDataSet` components. When data is provided to the data set, the `StorageDataSet` tracks the row status information (either deleted, inserted, or updated) for all rows. When data is *resolved* back to a data source like a SQL server, the row status information is used to determine which rows to add to, delete from, or modify in the SQL table. When a row has been successfully resolved, it obtains a new row status of resolved (either `RowStatus.UPDATE_RESOLVED`, `RowStatus.DELETE_RESOLVED`, or `RowStatus.INSERT_RESOLVED`). If the `StorageDataSet` is resolved again, previously resolved rows will be ignored, unless changes have been made subsequent to previous resolving. When data is *exported* to a text file, all of the data in the current view is written to the text file, and the row status information is not affected.

Data exported to a text file is sensitive to the current sorting and filtering criteria. If sort criteria are specified, the data is saved to the text file in the same order as specified in the sort criteria. If row order is important, remove the sort criteria prior to exporting data. If filter criteria are specified, only the data that meets the filter criteria will be saved. This is useful for saving subsets of data to different files, but could cause data loss if a filtered file is inadvertently saved over an existing data file.

Warning Remove filter criteria prior to saving, if you want to save all of the data back to the original file.

Exporting data from a QueryDataSet to a text file

Exporting data from a `QueryDataSet` to a text file is the same as exporting data from a `TableDataSet` component, as defined in [Chapter 16, “Tutorial: Importing and exporting data from a text file.”](#) JBuilder will create a SCHEMA file that defines each column, its name, and its data type so that the file can be imported back into JBuilder more easily.

Note BLOB columns are not exported, they are ignored when other fields are exported.

Saving changes from a TableDataSet to a SQL table

Use a `QueryResolver` to resolve changes back to a SQL table. For more information on using the `QueryResolver` to save changes to a SQL table, see [“Customizing the default resolver logic” on page 87](#).

Prior to resolving the changes back to the SQL table, you must set the table name and column names of the SQL table, as shown in the following code snippet. The SQL table and SCHEMA file must already exist. The applicable SCHEMA file of the `TableDataSet` must match the configuration of the SQL table. The variant data types of the `TableDataSet` columns must map to the JDBC types of server table. By default, all rows will have a status of INSERT.

```
tabledataset1.setTableName(string);
tableDataSet1.SetRowID(columnName);
```

Saving changes loaded from a `TextDataFile` to a JDBC data source

By default, data is loaded from a `TextDataFile` with a status of `RowStatus.Loaded`. Calling the `saveChanges()` method of a `QueryDataSet` or a `ProcedureDataSet` will not save changes made to a `TextDataFile` because these rows are not yet viewed as being inserted. To enable changes to be saved and enable all rows loaded from the `TextDataFile` to have an `INSERTED` status, set the property `TextDataFile.setLoadAsInserted(true)`. Now when the `saveChanges()` method of a `QueryDataSet` or a `ProcedureDataSet` is called, the data will be saved back to the data source.

For more information on using the `QueryResolver` to save changes to a SQL table, see [“Customizing the default resolver logic” on page 87](#).

Connecting to a database

To operate the database tutorials included in this book, you'll need to install the `JDataStore` JDBC driver. The InterClient JDBC driver can be used, as well. This section provides information for setting up `JDataStore` and InterClient for use in the tutorials.

Sun worked in conjunction with database and database tool vendors to create a DBMS independent API. Like ODBC (Microsoft's rough equivalent to JDBC), JDBC is based on the X/Open SQL Call Level Interface (CLI). Some of the differences between JDBC and ODBC are,

- JDBC is an all Java API that is truly cross platform. ODBC is a C language interface that must be implemented natively. Most implementations run only on Microsoft platforms.
- Most ODBC drivers require installation of a complex set of code modules and registry settings on client workstations. JDBC is an all Java implementation that can be executed directly from a local or centralized remote server. JDBC allows for much simpler maintenance and deployment than ODBC.

JDBC is endorsed by leading database, connectivity, and tools vendors including Oracle, Sybase, Informix, InterBase, DB2. Several vendors, including Borland, have JDBC drivers. Existing ODBC drivers can be utilized by way of the JDBC-ODBC bridge provided by Sun. Using the JDBC-ODBC bridge is not an ideal solution since it requires the installation of ODBC drivers and registry entries. ODBC drivers are also implemented natively which compromises cross-platform support and applet security.

JBuilder DataExpress components are implemented using the Sun database connectivity (JDBC) Application Programmer Interface (API). To create a Java data application, the Sun JDBC `sql` package must be accessible before you can start creating your data application. If your connection to your database server is through an ODBC driver, you also need the Sun JDBC-ODBC bridge software.

For more information about JDBC or the JDBC-ODBC bridge, visit the JDBC Database Access API web site at <http://java.sun.com/products/jdbc/>.

Connecting to databases

You can connect JBuilder applications to remote or local SQL databases, or to databases created with other Borland applications such as C++ Builder or Delphi.

To connect to a remote SQL database, you need either of the following:

- A JDBC driver for your server. Some versions of JBuilder include JDBC drivers. One of these drivers is InterClient. Check the Borland web site at <http://www.borland.com/jbuilder/> for availability of JDBC drivers in your edition of JBuilder or contact the technical support department of your server software company for availability of JDBC drivers.
- An ODBC-based driver for your server that you use with the JDBC-ODBC bridge software.

Note The ODBC driver is a non-portable DLL. This is sufficient for local development, but won't work for applets or other all-Java solutions.

When connecting to local, non-SQL databases such as Paradox or Visual dBASE, use an ODBC driver appropriate for the table type and level you are accessing in conjunction with the JDBC-ODBC bridge software.

Note When you no longer need a `Database` connection, you should explicitly call the `Database.closeConnection()` method in your application. This ensures that the JDBC connection is not held open when it is not needed and allows the JDBC connection instance to be garbage collected.

Adding a Database component to your application

The `Database` component is a JDBC-specific component that manages a JDBC connection. To access data using a `QueryDataSet` or a `ProcedureDataSet` component, you must set the `database` property of the component to an instantiated `Database` component. Multiple data sets can share the same database, and often will.

In a real world database application, you would probably place the `Database` component in a data module. Doing so allows all applications that access the database to have a common connection. To learn more about data modules, see [Chapter 10, "Using data modules to simplify data access."](#)

To add the `Database` component to your application,

- 1 Create a new project and application files using the Application wizard. (You can optionally follow these instructions to add data connectivity to an existing project and application.) To create a new project and application files:
 - a Choose `File|Close` from the JBuilder menu to close existing applications.
If you do not do this step before you do the next step, the new application files will be added to the existing project.
 - b Choose `File|New` and double-click the Application icon to start the Application wizard.
Accept or modify the default settings to suit your preferences.

- 2 Open the UI designer by selecting the Frame file (for example, `Frame1.java`) in the content pane, then select the Design tab at the bottom of the IDE.



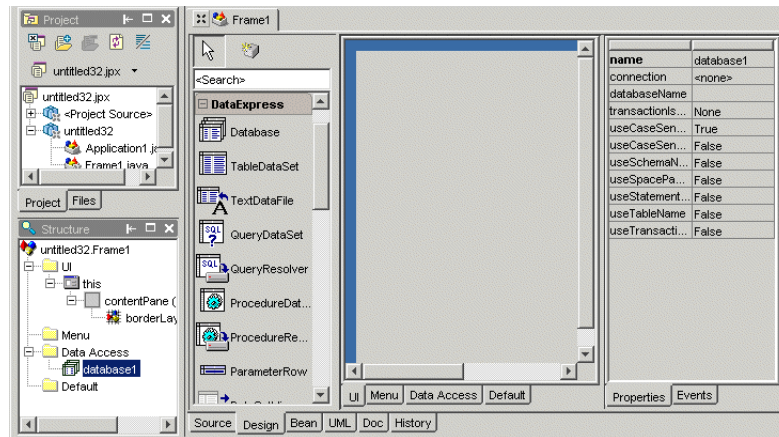
- 3 Select the DataExpress page on the component palette, and click the Database component.
- 4 Click anywhere in the designer window to add the Database component to your application.

This adds the following line of code to the Frame class:

```
Database database1 = new Database();
```

The Database component appears in the structure pane, looking like this:

Figure 4.1 Database component displayed in structure pane



Setting Database connection properties

The Database `connection` property specifies the JDBC driver, connection URL, user name, and password. The JDBC connection URL is the JDBC method for specifying the location of a JDBC data provider (for example, SQL server). It contains all the information necessary for making a successful connection, including user name and password.

You can access the `ConnectionDescriptor` object programmatically, or you can set connection properties through the Inspector. If you access the `ConnectionDescriptor` programmatically, follow these guidelines:

- If you set `promptPassword` to `true`, you should also call `openConnection()` for your database. `openConnection()` determines when the password dialog is displayed and when the database connection is made.
- Get user name and password information as soon as the application opens. To do this, call `openConnection()` at the end of the main frame's `jbInit()` method.

If you don't explicitly open the connection, it will try to open when a component or data set first needs data.

The following steps describe how to set connection properties to the sample JDataStore Employee database through the UI designer.

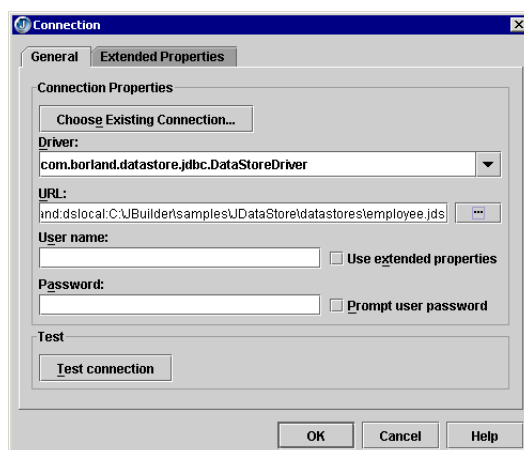
Note To use the sample database, you will need to make sure your system is set up for JDataStore. If you have not already done so, see [“Setting up JDataStore” on page 31](#).

- 1 Select `database1` in the component tree.
- 2 Select the `connection` property's value in the Inspector, and click the ellipsis (...) button to open the Connection property editor.
- 3 Set the following properties:

Property	Description
Driver	The class name of the JDBC driver that corresponds to the URL, for this example, select <code>com.borland.datastore.jdbc.DataStoreDriver</code> from the list.
URL	<p>The Universal Resource Locator (URL) of the database, for this example. The default value is <code>jdbc:borland:dslocal:directoryAndFile.jds</code>. Click the Browse button to select the Local JDataStore Database, which is located in <code><jbuilder>/samples/JDataStore/datastores/employee.jds</code>. Use the Browse button to select this file to reduce the chance of making a typing error. When selected, the URL will look similar to this:</p> <p>UNIX:</p> <pre>jdbc:borland:dslocal:/usr/local/<jbuilder>/samples/JDataStore/datastores/employee.jds</pre> <p>Windows:</p> <pre>jdbc:borland:dslocal:C:\jbuilder\samples\JDataStore\datastores\employee.jds</pre>
Username	The user name authorized to access the server database. For the sample tutorials, the user name is “SYSDBA”.
Password	The password for the authorized user. For the tutorials, the password is “masterkey”.
Prompt user for password	Whether to prompt the user for a password when opening the database connection.

The dialog looks like this:

Figure 4.2 Connection Descriptor dialog box



- 4 Click the Test Connection button to check that the connection properties have been correctly set.

The connection attempt results are displayed beside the Test Connection button.

If you cannot successfully connect to the sample database, make sure to set up your computer to use the JDataStore sample database. See [“Setting up JDataStore” on page 31](#) for more information.

- 5 Click OK to exit the Connection dialog box and write the connection properties to the source code when the connection is successful.

The source code, if the example above is followed, looks similar to this:

```
databasel.setConnection(new
    com.borland.dx.sql.dataset.ConnectionDescriptor(
        "jdbc:borland:dslocal:
        <jbuilder>/samples/JDataStore/datastores/employee.jds", "admin", "",
        false, "com.borland.datastore.jdbc.DataStoreDriver"));
```

- 6 Select a `DBDisposeMonitor` component from the More dbSwing page, and click in the content pane to add it to the application.

The `DBDisposeMonitor` will close the `JDataStore` when the window is closed.

- 7 Set the `dataAwareComponentContainer` property of the `DBDisposeMonitor` to this.

Tip Once a database URL connection is successful, you can use the Database Pilot to browse JDBC-based meta-database information and database schema objects in the `JDataStore`, and to execute SQL statements, and browse and edit data in existing tables.

Setting up JDataStore

To view and explore the contents of the `JDataStore`, use the `JDataStore Explorer`. To start the `JDataStore Explorer`, choose `Tools|JDataStore Explorer`. To open the sample `JDataStore`, browse to `<jbuilder>/samples/JDataStore/datastores/employee.jds`.

For more information on the `JDataStore Explorer`, see “`JDataStore Administration`” in the *JDataStore Developer's Guide*.

Setting up InterBase and InterClient

InterBase is a SQL-compliant, relational database management software product that is easy to use. *InterBase* is client and tools independent, supporting most of the popular desktop clients and application builder frameworks.

InterClient is an all-Java JDBC driver for *InterBase* databases. *InterClient* contains a library of Java classes which implement most of the JDBC API and a set of extensions to the JDBC API. It interacts with the JDBC Driver Manager to allow client-side Java applications and applets to interact with *InterBase* databases.

Note The current version of *InterClient* is a type 4 JDBC driver. Previous, type 3 drivers have been deprecated, and all development will continue for the type 4 driver only.

Developers can deploy InterClient-based clients in two ways:

- *Java applets* are Java programs that can be included in an HTML page with the `<APPLET>` tag, served by a web server, and viewed and used on a client system using a Java-enabled web browser. This deployment method doesn't require manual installation of the InterClient package on the client system. It does however require a Java-enabled browser on the client system.
- *Java applications* are stand-alone Java programs for execution on a client system. This deployment method requires the InterClient package, and the Java Runtime Environment (JRE) installed on the client system. The JRE includes the JDBC Driver Manager.

As an all-Java API to InterBase, InterClient enables platform-independent, client-server development for the Internet and corporate Intranets. The advantage of an all-Java driver versus a native-code driver is that you can deploy InterClient-based applets without having to manually load platform-specific JDBC drivers on each client system (the web servers automatically download the InterClient classes along with the applets). Therefore, there's no need to manage local native database libraries, which simplifies administration and maintenance of customer applications. As part of a Java applet, InterClient can be dynamically updated, further reducing the cost of application deployment and maintenance.

Using InterBase and InterClient with JBuilder

To use InterBase and InterClient with JBuilder, install InterBase and InterClient following the instructions in the InterBase documentation, then start the InterBase Server.

If you have trouble connecting, be sure the InterBase database is running. Database can run on the same machine as your application, or on a different machine. As a result, there are many possible configurations. It is important that your InterClient version be compatible with your database version and your JDK. For more information on these topics, please refer to the InterBase and InterClient documentation.

If InterBase Server is on a different platform than JBuilder, you need to:

- Make sure InterBase is running on the server.
- Make sure InterClient is installed on the client.
- Make sure the URL of the `Connection Descriptor` on the client has the correct IP address of the server running InterBase.

After InterClient is installed, add it to JBuilder using Enterprise|Enterprise Setup, then add it to your required list of libraries for your project in Project|Project Properties. For more information, see [“Adding a JDBC driver to JBuilder” on page 33](#).

Tips on using sample InterBase tables

For best results, note the following tips for working with the sample tables:

- Make a backup copy of the sample database.
Sample databases are installed by the setup program. You may wish to make a copy of the database file, `employee.gdb`, so that you can easily restore the file to its original condition after experimenting with database programming.

- Do not defy the database constraints.

The sample databases enforce many constraints on data values, as is normal in a realistic application. These constraints affect all examples where you add, insert, or

update data from the employee table and attempt to save the changes back to the server table.

- The EMPLOYEE table is used extensively in the examples in this manual. The following constraints apply to the employee table:
 - All fields are required (data must be entered) except for PHONE_EXT.
 - EMP_NO is generated, so no need to input for new records. It's also the primary key, so don't change it.
 - Referential integrity.
 - DEPT_NO must exist in Department table.
 - JOB_CODE, JOB_GRADE, JOB_COUNTRY must exist in JOB table.
 - SALARY must be greater than or equal to min_salary field from job table for the matching job_code, job_grade and job_country fields in job.
 - FULL_NAME is generated by the query so no need to enter anything.
- The CUSTOMER table is also used in the database tutorials. CUST_NO is generated, so there is no need to input for new records.

When working with the sample tables, it's safest to modify only the LAST_NAME, FIRST_NAME, PHONE_EXT fields in existing records.

To view the metadata for the sample tables,

- 1 Choose Tools|Database Pilot.

The Database Pilot is used for database administration tasks.

- 2 Double-click the database URL to open a connection to the database.
- 3 Expand the Tables node to view information about the individual sample tables.

Adding a JDBC driver to JBuilder

After installing your JDBC driver following the manufacturer's instructions, use the steps below to set it up for use with JBuilder.

Note Uninstalled drivers are red on the Drivers list in the Connection Property dialog box and cannot be selected for use in JBuilder. You must install them according to the manufacturer first before setting them up in JBuilder.

Creating the .library and .config files

There are three steps to adding a database driver to JBuilder:

- Creating a library file which contains the driver's classes, typically a JAR file, and any other auxiliary files such as documentation and source.
- Deriving a .config file from the library file which JBuilder adds to its classpath at start-up.
- Adding the new library to your project, or to the Default project if you want it available for all new projects.

The first two steps can be accomplished in one dialog box:

- 1 Open JBuilder and choose Enterprise|Enterprise Setup.
- 2 Click the Database Drivers tab in the Enterprise Setup dialog box.

The Database Drivers tab displays .config files for all the currently defined database drivers.

- 3 Click Add to add a new driver, then New to create a new library file for the driver. The library file is used to add the driver to the required libraries list for projects.
- Note** You can also create a new library under Tools|Configure Libraries, but since you would then have to use Enterprise Setup to derive the .config file, it is simpler to do it all here.
- 4 Type a name and select a location for the new file in the Create New Library dialog box.
- 5 Click Add, and browse to the location of the driver. You can select the directory containing the driver and all its support files, or you can select just the archive file for the driver. Either will work. JBuilder will extract the information it needs.
- 6 Click OK to close the file browser. This displays the new library at the bottom of the library list and selects it.
- 7 Click OK. JBuilder creates a new .library file in the JBuilder /lib directory with the name you specified (for example, `InterClient.library`). It also returns you to the Database Drivers page which displays the name of the corresponding .config file in the list which will be derived from the library file (for example, `InterClient.config`).
- 8 Select the new .config file in the database driver list and click OK. This places the .config file in the JBuilder /lib/ext directory.
- 9 Close and restart JBuilder so the changes to the database drivers will take effect, and the new driver will be put on the JBuilder classpath.

Important If you make changes to the .library file after the .config file has been derived, you must re-generate the .config file using Enterprise Setup, then restart JBuilder.

Adding the JDBC driver to projects

Projects run from within JBuilder use only the classpath defined for that project. Therefore, to make sure the JDBC driver is available for all new projects that will need it, define the library and add it to your default list of required libraries. This is done from within JBuilder using the following steps:

- 1 Start JBuilder and close any open projects.
- 2 Choose Project|Default Project Properties.
- 3 Select the Required Libraries tab on the Paths page, and click Add.
- 4 Select the new JDBC driver from the library list, and click OK.
- 5 Click OK to close the Default Project Properties dialog box.

Note You can also add the JDBC driver to an existing project. Just open the project, then choose Project|Project Properties and use the same process as above.

Now JBuilder and the new JDBC driver are set up to work together. The next step is to create or open a project that uses this driver, add a Database component to it, and set its `connection` property so it can use that driver to access the data. For an example of how to do this, see [“Connecting to a database using InterClient JDBC drivers” on page 35](#).

The `Database` component handles the JDBC connection to a SQL server and is required for all database applications involving server data. JDBC is the Sun Database Application Programmer Interface, a library of components and classes developed by Sun to access remote data sources. The components are collected in the `java.sql` package and represent a generic, low-level SQL database access framework.

The JDBC API defines Java classes to represent database connections, SQL statements, result sets, database metadata, etc. It allows a Java programmer to issue SQL statements and process the results. JDBC is the primary API for database access in Java. The JDBC API is implemented via a driver manager that can support multiple drivers connecting to different databases. For more information about JDBC, visit the Sun JDBC Database Access API web site at <http://java.sun.com/products/jdbc/>.

JBuilder uses the JDBC API to access the information stored in databases. Many of JBuilder's data-access components and classes use the JDBC API. Therefore, these classes must be properly installed in order to use the JBuilder database connectivity components. In addition, you need an appropriate JDBC driver to connect your database application to a remote server. Drivers can be grouped into two main categories: drivers implemented using native methods that bridge to existing database access libraries, or all-Java based drivers. Drivers that are not all-Java must run on the client (local) system. All-Java based drivers can be loaded from the server or locally. The advantages to using a driver entirely written in Java are that it can be downloaded as part of an applet and is cross-platform.

Some versions of JBuilder include JDBC drivers. Check the Borland web site at <http://www.borland.com/jbuilder/> for availability of JDBC drivers in the JBuilder versions, or contact the technical support department of your server software company for availability of JDBC drivers. Some of the driver options that may ship with JBuilder are:

- DataStoreDriver

DataStoreDriver is the JDBC driver for the `JDataStore` database. The driver supports both local and remote access. Both types of access require a user name.

Note

The user name can be any string. If no user has been added to the store's user table, there is no need to provide one. For transactional stores, however, a not-null string is required for user name.

For instructions on connecting to a database using the `JDataStore` driver, see [Chapter 17, "Tutorial: Creating a basic database application."](#)

- InterClient

InterClient is a JDBC driver that you can use to connect to InterBase. InterClient can be installed by running the InterClient installation program. Once installed, InterClient can access InterBase sample data using the `ConnectionDescriptor`.

For information on connecting to a database using InterClient, see ["Connecting to a database using InterClient JDBC drivers" on page 35](#).

You can connect JBuilder applications to remote or local SQL databases, or to databases created with other Borland applications such as C++ Builder or Delphi. To do so, look at the underlying database that your application connects to and connect to that database using its connection URL.

Connecting to a database using InterClient JDBC drivers

This section discusses adding a `Database` component, which is a JDBC-specific component that manages a JDBC connection, and setting the properties of this component that enable you to access sample InterBase data.

In a real world database application, you would probably place the `Database` component in a data module. Doing so allows all applications that access the database to have a common connection. To learn more about data modules, see [Chapter 10, "Using data modules to simplify data access."](#)

To add the `Database` component to your application and connect to the InterBase sample files,

- 1 Make sure to follow the steps in “Setting up InterBase and InterClient” on page 31 and “Adding a JDBC driver to JBuilder” on page 33 to make sure your system is correctly set up for accessing the sample InterBase files.
- 2 Close all projects and create a new application, or add data connectivity to an existing project and application.

You can create a new project and application files by choosing File|New, and double-clicking the Application icon. Select all defaults. JBuilder will create the necessary files and display them in the project pane. The file `Frame1.java` will be open in the content pane. `Frame1.java` will contain the user interface components for this application.

- 3 Click the Design tab on `Frame1.java` at the bottom of content pane.



- 4 Select the DataExpress page on the component palette, and click the `Database` component.
- 5 Click anywhere in the content pane or UI designer to add the `Database` component to your application.
- 6 Set the `Database connection` property to specify the JDBC driver, connection URL, user name, and password.

The JDBC connection URL is the JDBC method for specifying the location of a JDBC data provider (i.e., SQL server). It can actually contain all the information necessary for making a successful connection, including user name and password.

To set the `connection` property,

- a Make sure the `Database` object is selected in the content pane.

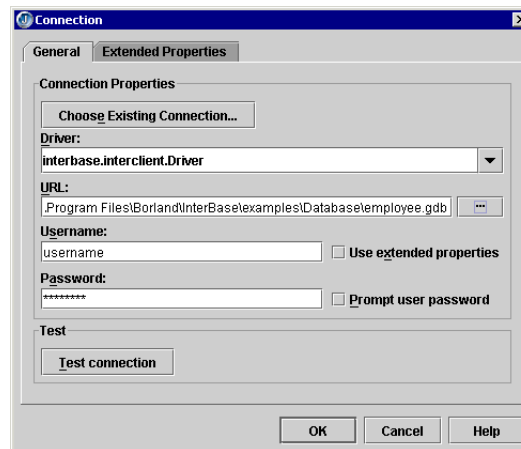
Double-click the `connection` property in the Inspector to open the `connection` property editor. In this example, the data resides on a Local InterBase server. If your data resides on a remote server, you would type the IP address of the server instead of “localhost” entered here.

- b Set the following properties:

Property	Value
Driver	<code>interbase.interclient.Driver</code>
URL	<p>Browse to the sample InterBase file, <code>employee.gdb</code>, located in your InterBase <code>/examples</code> directory. The entry in the URL field will look similar to this:</p> <p>UNIX:</p> <pre>jdbc:interbase://localhost//usr/interbase/examples/employee.gdb</pre> <p>Windows:</p> <pre>jdbc:interbase://localhost/D:\InterBaseCorp\InterBase\examples\database\employee.gdb</pre>
Username	<code>SYSDBA</code>
Password	<code>masterkey</code>

The dialog box looks like this:

Figure 4.3 Connection Descriptor dialog



- c Click the Test Connection button to check that the connection properties have been correctly set.

The connection attempt results are displayed directly beneath the Test Connection button. See [“Common connection error messages” on page 43](#) for solutions to some typical connection problems.

- d Click OK to exit the dialog and write the connection properties to the source code when the connection is successful.

Tip Once a database URL connection is successful, you can use the Database Pilot to browse JDBC-based meta-database information and database schema objects, as well as execute SQL statements, and browse and edit data in existing tables.

Using the Database component in your application

Now that your application includes the `Database` component, you’ll want to add another `DataExpress` component that retrieves data from the data source to which you are connected. JBuilder uses queries and stored procedures to return a set of data. The components implemented for this purpose are `QueryDataSet` and `ProcedureDataSet`. These components work with the `Database` component to access the SQL server database. For instructions on how to use these components, see the following sections:

- [“Querying a database” on page 46](#)
- [“Using parameterized queries to obtain data from your database” on page 53](#)
- [“Using stored procedures” on page 61](#)

Most of the sample applications and tutorials use a `Database` connection to the sample EMPLOYEE `JDataStore`, as described here.

For most database applications, you would probably encapsulate the `Database` and other `DataExpress` components in a `DataModule` instead of directly adding them to an application’s `Frame`. For more information on using the `DataExpress` package’s `DataModule`, see [Chapter 10, “Using data modules to simplify data access.”](#)

Prompting for user name and password

When developing a database application, it is convenient to include a user name and password in the `ConnectionDescriptor` so that you do not have to supply this information each time you use the designer or run your application. If you set the `ConnectionDescriptor` through the designer, the designer writes the code for you. Before you deploy your application, you will probably want to remove the user name and password from the code, prompting the user for the information at runtime instead, particularly if you distribute the source code or if different users have different access rights. You have several options for prompting a user for their user name and password at runtime.

- Check the Prompt User For Password checkbox in the editor for the `Database connection` property, or write code to set the `ConnectionDescriptor` `promptPassword` parameter to `true`.

At runtime and when you show live data in the Designer, a user name and password dialog will display. A valid user name and password must be entered before data will display.

- Add an instance of `dbSwing DBPasswordPrompter` to your application.

This option gives you more control over user name/password handling. You can specify what information is required (only user name, only password, or both), how many times the user can attempt to enter the information, and other properties. The OK button will be disabled until the necessary information is supplied. The dialog is displayed when you call its `showDialog()` method. This allows you to control when it appears. Be sure to present it early in your application, before any visual component tries to open your database and display data. The designer doesn't call `showDialog()`, so you need to specify user name and password in the `ConnectionDescriptor` when you activate the designer.

Pooling JDBC connections

For applications which require many database connections, you should consider connection pooling. Connection pooling provides significant performance gains, especially in cases where large numbers of database connections are opened and closed.

`JDataStore` provides several components for dealing with JDBC 2.0 `DataSources` and connection pooling. Use of these components requires J2EE. If your version of `JBuilder` does not include `J2EE.jar`, you will need to download it from Sun, and add it to your project as a required library. See [“Adding a required library to a project” on page 106](#) for instructions on adding a required library.

The basic idea behind connection pooling is simple. In an application that opens and closes many database connections, it is efficient to keep unused `Connection` objects in a pool for future re-use. This saves the overhead of having to open a new physical connection each time a connection is opened.

Here are the main `DataSource` and connection pooling components provided by `JDataStore`:

- `JDBCDataSource` is an implementation of the `javax.sql.DataSource` interface. `JDBCDataSource` can create a connection to a `JDataStore`, or any other JDBC driver, based on its JDBC connection properties, but it does no connection pooling. Because it is an implementation of `javax.sql.DataSource`, it can be registered with a JNDI naming service. For information on JNDI naming services, consult the JDK documentation, or <http://www.javasoft.com>.
- `JDBCConnectionPool` is also an implementation of `javax.sql.DataSource`, and therefore can be registered with a JNDI naming service. `JDBCConnectionPool` can be used to provide connection pooling with any JDBC driver. It creates connections based on its JDBC connection properties. `JDBCConnectionPool` has various properties for

connection pool management, for instance, properties specifying a minimum and maximum number of connections.

When using `JdbcConnectionPool`, you are required to set the `connectionFactory` property. This allows `JdbcConnectionPool` to create `javax.sql.PooledConnection` objects. The `connectionFactory` property setting must refer to an implementation of `javax.sql.ConnectionPoolDataSource` (such as `JdbcConnectionFactory`). The `connectionFactory` property can also be set by using the `dataSourceName` property. The `dataSourceName` property takes a `String`, which it will look up in the JNDI naming service to acquire the implementation of `javax.sql.ConnectionPoolDataSource`.

To get a connection from the pool, you will usually call `JdbcConnectionPool.getConnection()`. The connection returned by this method does not support distributed transactions, but it can work with any JDBC driver.

`JDBCConnectionPool` also provides support for distributed transactions (XA), but this feature is only available when `JDBCConnectionPool` is used in conjunction with the `JDataStore` JDBC driver, and is only useful when combined with a distributed transaction manager, such as the Borland Enterprise Server. For more information on `JDBCConnectionPool`'s XA support, see "Connection pooling and distributed transaction support" in the *JDataStore Developer's Guide*.

- `JdbcConnectionFactory` is an implementation of `javax.sql.ConnectionPoolDataSource`. It is used to create `javax.sql.PooledConnection` objects for a connection pool implementation like `JDBCConnectionPool`.

`JDBCConnectionPool` and `JDBCConnectionFactory` are easily used together, but they can also each be used separately. The decoupling of these two components provides more flexibility. For example, `JDBCConnectionFactory` could be used with another connection pooling component which uses a different strategy than `JDBCConnectionPool`. `JDBCConnectionFactory` can be used with any JDBC 2.0 connection pool implementation that allows a `javax.sql.ConnectionPoolDataSource` implementation (like `JDBCConnectionFactory`) to provide its `javax.sql.PooledConnections`.

`JDBCConnectionPool`'s efficient pooling strategy, on the other hand, could be used with another connection factory implementation. `JDBCConnectionPool` can be used with any JDBC driver that provides a connection factory component which implements `javax.sql.ConnectionPoolDataSource`.

Here are the main `DataSource` and connection pooling components provided by InterClient for InterBase databases:

- `interbase.interclient.DataSource` is an implementation of the `javax.sql.DataSource` interface. `DataSource` can create a connection to an InterBase database, based on its JDBC connection properties, but it does no connection pooling. Because it is an implementation of `javax.sql.DataSource`, it can be registered with a JNDI naming service. For information on JNDI naming services, consult the JDK documentation, or <http://www.javasoft.com>.
- `interbase.interclient.ConnectionPoolModule` is also an implementation of `javax.sql.DataSource`, and therefore can be registered with a JNDI naming service. `ConnectionPoolModule` can be used to provide connection pooling with any JDBC driver. It creates connections based on its JDBC connection properties. `ConnectionPoolModule` has various properties for connection pool management, such as properties specifying a minimum and maximum number of connections. When using `ConnectionPoolModule`, you are required to set the `connectionFactory` property. To get a connection from the pool, you will usually call `ConnectionPoolModule.getConnection()`. The connection returned by this method and InterClient does not support distributed transactions.
- `interbase.interclient.JdbcConnectionFactory` is an implementation of `javax.sql.ConnectionPoolDataSource`. It is used to create `javax.sql.PooledConnection` objects for a connection pool implementation like `ConnectionPoolModule`.

Now that we've given you an overview of the classes involved in connection pooling, it's time to explain a bit more about how they work:

- The `JdbcConnectionPool.getConnection()` method tries to save the overhead of opening a new connection by using a connection that is already in the pool. When a lookup is performed to find a connection in the pool, a match is found if the user name equals the user name that was originally used to create the pooled connection. Password is not considered when trying to match a user. A new connection is requested from the factory only if no match is found in the pool.
- Connection pooling is a relatively simple, but very powerful API. Most of the difficult work, like keeping track of pooled connections, and deciding whether to use an existing pooled connection or open a new one, is done completely internally.
- When an application uses connection pooling, a connection should always be explicitly closed when no longer in use. This allows the connection to be returned to the pool for later use, which improves performance.
- The factory which creates connections for the pool should use the same property settings for all of them, except for the user name and password. A connection pool, therefore, accesses one database, and all its connections have the same JDBC connection property settings (but can have different usernames/passwords).

Optimizing performance of JConnectionPool

The lookup mechanism for finding a pooled connection that shares the same user name does a quick scan comparing user name string references. If possible, pass in the same `String` instance for all connection requests. One way to ensure this is to always use a constant name specified as follows for connection pooling:

```
public static final String POOL_USER = "CUSTOMER_POOL_USER";
```

Logging output

Both `JdbcConnectionPool` and `JdbcConnectionFactory` have `PrintWriter` properties. Most log output has the form of:

```
[<class instance hashCode>]:<class name>.<method name>(...)
```

Any hexadecimal values displayed in square brackets (`[]`) in the log files are `hashCode()` values for an `Object`.

Pooling example

The following is a trivial example of using connection pooling. This data module code fragment shows the most important and most basic lines of code you will need in an application using connection pooling, without making too many assumptions about what your specific application may need to do with this technology. For a non-trivial example of connection pooling, refer to the Web Bench sample in `samples/JDataStore/WebBench`. For more information about data modules, see [Chapter 10, "Using data modules to simplify data access."](#)

```
import com.borland.dx.dataset.*;
import com.borland.dx.sql.dataset.*;
import com.borland.javax.sql.*;
import java.sql.*;

public class DataModule1 implements DataModule {

    private static DataModule1 myDM;
    private static final String POOL_USER = "POOL_USER";
```



```

private static JdbcConnectionFactory factory;
private static JdbcConnectionPool pool;

public DataModule1() {
    try {
        jbInit();
    }
    catch(Exception e) {
        e.printStackTrace();
    }
}

private void jbInit() throws Exception {
    // Instantiate connection factory
    factory = new JdbcConnectionFactory();

    // The next line sets the URL to a
    // local JDataStore file. The specific
    // URL will depend on the location
    // of your JDataStore file.
    factory.setUrl("jdbc:borland:dslocal:<path><file name>");
    factory.setUser(POOL_USER);
    factory.setPassword("");

    // Instantiate the connection pool
    pool = new JdbcConnectionPool();
    // Assign the connection factory as
    // the factory for this pool
    pool.setConnectionFactory(factory);
}

public Connection getConnection() {

    Connection con = null;

    try {
        con = pool.getConnection();
    } catch (Exception ex) {
        ex.printStackTrace();
    }

    return con;
}

public static DataModule1 getDataModule() {
    if (myDM == null) {
        myDM = new DataModule1();
    }
    return myDM;
}

public static JdbcConnectionPool getPool() {
    return pool;
}
}

```

You will probably write the code for the application logic in a separate source file. The next code fragment shows how to request connections from the pool, and later, how to

make sure the connections are returned to the pool. It also shows how to make sure the pool is shut down when the application ends.

```
public class doSomething {

    static DataModule1 dm = null;

    public doSomething() {
    }

    public static void main(String args[]){

        // Several of the methods called here could throw exceptions,
        // so exception handling is necessary.

        try{
            // Instantiate the data module
            dm = new DataModule1();
            java.sql.Connection con = null;

            // This application gets 100 connections
            // and returns them to the pool.
            for (int i=0; i<100; i++){

                try{
                    // Get a connection
                    con = dm.getPool().getConnection();
                }

                catch(Exception e){
                    e.printStackTrace();
                }

                finally{
                    // Return the connection to the pool
                    con.close();
                }
            }
        }
        catch(Exception x){
            x.printStackTrace();
        }
        finally{
            try{
                // Shut down the pool before the
                // the application exits.
                dm.getPool().shutdown();
            }
            catch(Exception ex){
                ex.printStackTrace();
            }
        }
    }
}
```

Troubleshooting JDataStore and InterBase connections

Connecting to a SQL server using JDBC can result in error messages generated by JDBC. For help with troubleshooting JDataStore connections in the tutorials,

- Read “Troubleshooting” in the *JDataStore Developer’s Guide*.
- Check the Borland JDataStore FAQ at <http://community.borland.com/article/0,1410,19685,00.html>.
- Read “[Common connection error messages](#)” on [page 43](#) for problems connecting to InterBase via InterClient.

Common connection error messages

Listed below are some common connection errors and solutions:

- Unable to locate the InterClient driver. InterClient has not been added as a required library for the project. Choose Project|Project Properties, and add InterClient as a Required Library.
- Driver could not be loaded. InterClient has not been added to the CLASSPATH. Add the `interclient.jar` file to the JBuilder startup script CLASSPATH, or to your environment’s CLASSPATH before launching JBuilder.

Retrieving data from a data source

This chapter focuses on using JBuilder's DataExpress architecture to retrieve data from a data source, and provide data to an application. The components in the DataExpress packages encapsulate both the connection between the application and its source of the data, as well as the behavior needed to manipulate the data.

To create a database application, you retrieve information stored in the data source, and create a copy that your application can manipulate locally. The data retrieved from the provider is cached inside a `DataSet`. All changes to the cached `DataSet` are tracked so that resolver implementations know what needs to be inserted, updated, or deleted back to the data source. In JBuilder, a subset of data is extracted from the data source into a JBuilder `StorageDataSet` subclass. The `StorageDataSet` subclass you use depends on the way in which you obtain the information.

Using a provider/resolver approach, you only need two interactions between the database application and the data source: the initial connection to retrieve the data, and the final connection to resolve the changes back to the data source. The connection between the `DataSet` component client and the data source can be disconnected after data is provided, and only needs to be re-established for the duration of the resolving transaction.

DataExpress components also provide support for direct data binding to dbSwing components. You simply set a property in the Inspector to bind Data to visual components.

Some of the examples in this chapter use a `JDataStore` driver to access data in a `JDataStore`. Others use a JDBC driver to access data in InterBase tables. Both of these options have their advantages. Which you choose depends on your application needs. With both options,

- You can directly wire visual components.
- You get full featured data access that includes master-detail, sorting, filtering, and constraints.
- You can track edits to retrieved data so they can be correctly resolved to the data source.

Querying a database

A `QueryDataSet` component is a JDBC-specific `DataSet` that manages a JDBC data provider, as defined in the `query` property. You can use a `QueryDataSet` component in JBuilder to extract data from a data source into a `StorageDataSet` component. This action is called *providing*. Once the data is provided, you can view and work with the data locally in data-aware components. When you want to save the changes back to your database, you must resolve the data. The DataExpress architecture is discussed in more detail in [Chapter 2, “Understanding JBuilder database applications.”](#)

`QueryDataSet` components enable you to use SQL statements to access, or provide, data from your database. You can add a `QueryDataSet` component directly to your application, or add it to a data module to centralize data access and control business logic.

To query a SQL table, you need the following components, which can be supplied programmatically or with JBuilder design tools:

- **Database**
The `Database` component encapsulates a database connection through JDBC to the SQL server and also provides lightweight transaction support.
- **QueryDataSet**
A `QueryDataSet` component provides the functionality to run a query statement (with or without parameters) against tables in a SQL database, and stores the result set from the execution of the query.
- **QueryDescriptor**
The `QueryDescriptor` object stores the query properties, including the database to be queried, the query string to execute, and optional query parameters.

The `QueryDataSet` has built-in functionality to fetch data from a JDBC data source. However, the built-in functionality (in the form of the default resolver) does much more than fetch data. It also generates the appropriate SQL INSERT, UPDATE, and DELETE queries for saving changes back to the data source after it has been fetched.

The following properties of the `QueryDescriptor` object affect query execution. These properties can be set visually in the `query` property editor. For a discussion of the `query` property editor and its tools and properties, see [“Setting properties in the query dialog box” on page 47.](#)

Property	Effect
database	Specifies what <code>Database</code> connection object to run the query against.
query	A SQL statement (typically a SELECT statement).
parameters	An optional <code>ReadWriteRow</code> from which to fill in parameters, used for parameterized queries.
executeOnOpen	Causes the <code>QueryDataSet</code> to execute the query when it is first opened. This is useful for presenting live data at design time. You may also want this enabled at run time.
loadOption	An optional integer value that defines the method of loading data into the data set. Options are: <ul style="list-style-type: none"> ■ Load All Rows: load all data up front. ■ Load Rows Asynchronously: causes the fetching of <code>DataSet</code> rows to be performed on a separate thread. This allows the <code>DataSet</code> data to be accessed and displayed as the <code>QueryDataSet</code> is fetching rows from the database connection. ■ Load As Needed: load the rows as they are needed. ■ Load One Row At A Time: load as needed and replace the previous row with the current. Useful for high-volume batch-processing applications.

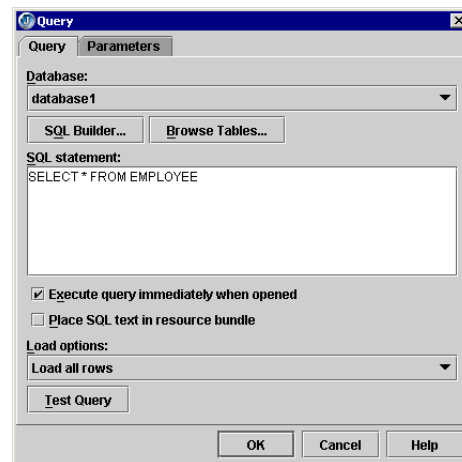
A `QueryDataSet` can be used in three different ways to fetch data.

- **Unparameterized queries:** The query is executed and rows are fetched into the `QueryDataSet`.
- **Parameterized queries:** You use variables in the SQL statement and then supply the actual parameters to fill in those values. For more information on parameterized queries, see [“Using parameterized queries to obtain data from your database” on page 53](#).
- **Dynamic fetching of detail groups:** Records from a detail data set are fetched on demand and stored in the detail data set. For more information, see [“Fetching details” on page 97](#).

Setting properties in the query dialog box

The Query property editor displays when you click the ellipsis button in the value field for the `query` property of a `QueryDataSet`. You can use the Query property editor to set the properties of the `QueryDescriptor` visually, but it also has several other uses. The Query property editor is shown below. Each of its options is explained in further detail as well.

Figure 5.1 Query property editor



For more information, see the `com.borland.dx.sql.dataset.QueryDescriptor` topic in the *DataExpress Component Library Reference* documentation.

The Query page

On the Query tab, the following options are available:

- The **Database** drop-down list displays the names of all instantiated `Database` objects to which this `QueryDataSet` can be bound. This property must be set for the query to run successfully. To instantiate a `Database`, see [Chapter 4, “Connecting to a database.”](#)

Selecting a `Database` object enables the SQL Builder and Browse Tables button.

- Click the **SQL Builder** button to display the SQL Builder. The SQL Builder provides a visual representation of the database, and allows you to create a SQL Statement by selecting Columns, adding a Where clause, an Order By clause, a Group By clause, and viewing and testing the generated SQL Statement. When you click OK, the SQL Statement you created with the SQL Builder will be placed in the SQL Statement field of the Query dialog.
- Click the **Browse Tables** button to display the Available Tables and Columns dialog. The Available Tables and Columns dialog displays a list of tables in the

specified `Database`, and the columns in the selected table. The Paste Table and Paste Column buttons allow you to quickly create your query statement by pasting the name of the selected table (by clicking the Paste Table button) or selected column (by clicking the Paste Column button) into your query statement at the cursor's current (insertion) point.

This button is dimmed and unavailable while the Database field displays the value "<none>". Select a `Database` object in the Database field to enable this button.

- **SQL Statement** is a Java String representation of a SQL statement (typically a SELECT statement). Enter the query statement to run against the `Database` specified in the Database drop-down list. Use the Browse Tables button to quickly paste the selected table and column names into the query statement. This is a required property; you must specify a valid SQL statement. If the SQL statement does not return a result set, an exception is generated.

An example of a simple SQL statement that is used throughout this text selects three fields from the EMPLOYEE table:

```
SELECT emp_no, last_name, salary FROM employee
```

This following SQL statement selects all fields from the same table.

```
SELECT * FROM employee
```

- The **Execute Query Immediately When Opened** option determines whether the query executes automatically when the `QueryDataSet` is opened. This option defaults to checked, which allows live data to display in the UI designer when the `QueryDataSet` is bound to a data-aware component.
- **Load Options** are optional integer values that define the method of loading data into the data set. Options are:
 - a Load All Rows: load all data up front.
 - b Load Rows Asynchronously: causes the fetching of `DataSet` rows to be performed on a separate thread. This allows the `DataSet` data to be accessed and displayed as the `QueryDataSet` is fetching rows from the database connection.
 - c Load As Needed: load the rows as they are needed.
 - d Load One Row At A Time: load as needed and replace the previous row with the current. Useful for high-volume batch-processing applications.
- When **Place SQL Text In Resource Bundle** is checked, upon exiting the `query` property editor, the Create ResourceBundle dialog displays. Select a resource bundle type. When the OK button is clicked, the SQL text is written to a resource file so that you can continue to use source code to persist SQL for some applications. See ["Place SQL text in resource bundle" on page 49](#) for more description of this feature.

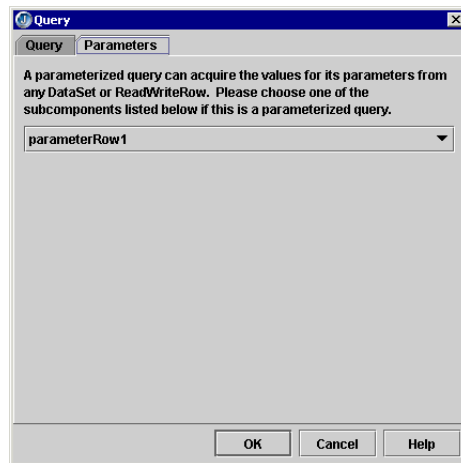
If unchecked, the SQL string is written to the `QueryDescriptor` as a String embedded in the source code.

- Click **Test Query** to test the SQL statement and other properties on this dialog against the specified `Database`. The result ("Success" or "Fail") is displayed in the gray area directly beneath the Test Query button. If the area below the button indicates success, the query will run. If it indicates Fail, review the information you have entered in the `query` for spelling and omission errors.

The Parameters page

On the Parameters tab, you can select an optional `ReadWriteRow` or `DataSet` from which to fill in parameters, used for parameterized queries. Parameter values are specified through an instantiated `ReadWriteRow`. Select the `ReadWriteRow` object (or the `ReadWriteRow` subclass) that contains the values for your query parameters from the drop-down list.

Figure 5.2 Parameters page



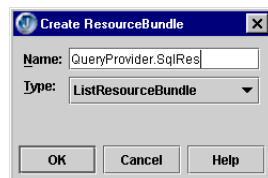
Any `ReadWriteRow`, such as `ParameterRow`, `DataSet`, and `DataRow` may be used as query or procedure parameters. In a `ParameterRow`, columns can simply be set up with the `addColumnns` and `setColumns` methods. `DataSet` and `DataRow` should only be used if they already contain the columns with the wanted data. See [“Using parameterized queries to obtain data from your database” on page 53](#) for an example of this.

Place SQL text in resource bundle

A `java.util.ResourceBundle` contains locale-specific objects. When your program needs a locale-specific resource, your program can load it from the resource bundle that is appropriate for the current user’s locale. In this way, you can write program code that is largely independent of the user’s locale isolating most, if not all, of the locale-specific information in resource bundles.

The Create `ResourceBundle` dialog appears when the `query` editor is closing, if a SQL statement has been defined in the `query` editor and the “Place SQL Text In Resource Bundle” option has been checked. The resource bundle dialog looks like this:

Figure 5.3 Resource Bundle dialog



To use a resource bundle in your application,

- 1 Select a type of `ResourceBundle`.

To simplify things, the JDK provides two useful subclasses of `ResourceBundle`: `ListResourceBundle` and `PropertyResourceBundle`. The `ResourceBundle` class is itself an abstract class. In order to create a concrete bundle, you need to derive from `ResourceBundle` and provide concrete implementations of some functions which retrieve from whatever storage you put your resources in, such as string. You can store resources into this bundle by right-clicking a property and specifying the key. JBuilder will write the strings into the resource file in the right format depending on the type.

- If you select `ListResourceBundle`, a Java file will be generated and added to the project. With `ListResourceBundle`, the messages (or other resources) are stored in a 2-D array in a Java source file. `ListResourceBundle` is again an abstract class. To create an actual bundle that can be loaded, you derive from `ListResourceBundle`

and implement `getContents()`, which most likely will just point to a 2D array of key-object pairs. For the above example you would create a class:

```
package myPackage;
public class myResource extends ListResourceBundle {
    Object[][] contents = {
        {"Hello_Message", "Howdy mate"}
    }
    public Object[][] getContents() {
        return contents;
    }
}
```

- If you select `PropertyResourceBundle`, a properties file will be created. The `PropertyResourceBundle` is a concrete class, which means you don't need to create another class in order to use it. For property resource bundles, the storage for the resources is in files with a `.properties` extension. When implementing a resource bundle of this form, you simply provide a properties file with the right name and store it in the same location as the class files for that package. For the above example, you would create a file `myResource.properties` and put it either in the CLASSPATH or in the zip/jar file, along with other classes of the `myPackage` package. This form of resource bundle can only store strings, and loads a lot slower than class-based implementations like `ListResourceBundle`. However, they are very popular because they don't involve working with source code, and don't require a recompile. The contents of the properties file will be like this:

```
# comments
Hello_message=Howdy mate
```

2 Click Cancel or OK:

Clicking the Cancel button (or deselecting the "Place SQL text in resource bundle" option in the query dialog), writes a `QueryDescriptor` like the following to the Frame file. The SQL text is written as a string embedded in the source code.

```
queryDataSet1.setQuery(new
    com.borland.dx.sql.dataset.QueryDescriptor(database1,
        "select * from employee", null, true, LOAD.ALL));
```

Clicking the OK button creates a `queryDescriptor` like the following:

```
queryDataSet1.setQuery(new
    com.borland.dx.sql.dataset.QueryDescriptor(database1,
        sqlRes.getString("employee"), null, true, LOAD.ALL));
```

Whenever you save the SQL text in the `QueryDescriptor` dialog, JBuilder automatically creates a new file called `SqlRes.java`. It places the text for the SQL string inside `SqlRes.java` and creates a unique string tag which it inserts into the text. For example, for the select statement `SELECT * FROM employee`, as entered above, the moment the OK is clicked, the file `SqlRes.java` would be created, looking something like this:

```
public class SqlRes extends java.util.ListResourceBundle {
    static final Object[][] contents = {
        { "employee", "select * from employee" }};
    static final java.util.ResourceBundle res = getBundle("untitled3.SqlRes");
    public static final String getStringResource(String key) {
        return res.getString(key);
    }
    public Object[][] getContents() {
        return contents;
    }
}
```

If the SQL statement is changed, the changes are saved into `SqlRes.java`. No changes will be necessary to the code inside `jbInit()`, because the "tag" string is invariant.

For more information on resource bundles, see the JavaDoc for `java.util.ResourceBundle`, found from JBuilder help by choosing HelpJava Reference. Then select the `java.util` package, and the `ResourceBundle` class.

Querying a database: Hints & tips

This set of topics includes tips to help you

- Enhance data set performance
- Open and close data sets most efficiently
- Ensure that a query is updatable

Enhancing data set performance

This section provides some tips for fine-tuning the performance of a `QueryDataSet` and a `QueryProvider`. For enhancing performance during data retrieval, eliminate the query analysis that the `QueryProvider` performs by default when a query is executed for the first time. See [“Persisting query metadata” on page 52](#) for information on doing this.

- Set the `loadOption` property on the `Query/ProcedureDataSet` components to `Load.ASYNCHRONOUS` or `Load.AS_NEEDED`. You can also set this property to `Load.UNCACHED` if you will be reading the data one time, and in sequential order.
- For large result sets, use a `JDataStore` to improve performance. With this option, the data is saved to disk rather than to memory.
- Cache SQL statements. By default, DataExpress will cache prepared statements for both queries and stored procedures if `java.sql.Connection.getMetaData().getMaxStatements()` returns a value greater than 10. You can force statement caching in JBuilder by calling `Database.setUseStatementCaching(true)`.

The prepared statements that are cached are not closed until one of the following happens:

- Some provider related property, like the `query` property, is changed.
- A `DataSet` component is garbage collected (statement closed in a `finalize()` method).
- `QueryDataSet.closeStatement()`, `ProcedureDataSet.closeStatement()`, `QueryProvider.closeStatement()`, or `ProcedureProvider.closeStatement()` is called.

To enhance performance during data inserts/deletes/updates:

- For updates and deletes,
 - a Set the `Resolver` property to a `QueryResolver`.
 - b Set the `UpdateMode` property of this `QueryResolver` to `UpdateMode.KEY_COLUMNS`.

These actions weaken the optimistic concurrency used, but reduce the number of parameters set for an update/delete operation.

- Set the `useTransactions` property of your `Database` to `false`. This property is `true` by default if the database supports transactions. When it is `true`, each insert, delete, or update statement is treated as a separate, automatically-committed transaction. When you set `useTransactions` to `false`, the statements are all processed in a single transaction.

Note In this case, you must call the `Database` or the `commit()` method of the `Connection` to complete the transaction (or call `rollback()` to discard all the changes).

- Disable the `resetPendingStatus` flag in the `Database.saveChanges()` method to achieve further performance benefits. With this disabled, DataExpress will not clear the `RowStatus` state for all inserted/deleted/updated rows. This is only desirable if you will not be calling `saveChanges()` with new edits on the `DataSet` without calling `refresh` first.

Persisting query metadata

By default, a query is analyzed for updatability the first time it is executed. This analysis involves parsing the query string and calling several methods of the JDBC driver. This analysis is potentially very expensive. You can remove the time overhead from run time, however, and perform the analysis during design of a form or data model.

To do this,



- 1 Highlight the `QueryDataSet` in the designer, and double-click it.
- 2 Click the Persist All Metadata button in the column designer.

The query is now analyzed, and a set of property settings will be added to the code. For more discussion of the Persist All Metadata button, see [“Using the column designer to persist metadata” on page 71](#). To set the properties without using the designer,

- 1 Set the `metaUpdate` property for the `StorageDataSet` to `NONE`.
- 2 Set the `tableName` property for the `StorageDataSet` to the table name for single table queries.
- 3 Set the `rowID` property of `Column` for the columns so that they uniquely and efficiently identify a row.
- 4 Change the query string to include columns that are suitable for identifying a row (see previous bullet), if not already included. Such columns should be marked invisible with the `visible` or `hidden` property for the `Column`.
- 5 Set the column properties `precision`, `scale`, and `searchable` to appropriate values. These properties are not needed if the `metaDataUpdate` property is in something other than `NONE`.
- 6 Set the `tableName` property of `Column` for multi-table queries.
- 7 Set the `serverColumnName` property of `Column` to the name of the column in the corresponding physical table if an alias is used for a column in the query.

Opening and closing data sets

`Database` and `DataSet` are implicitly opened when components bound to them open. When you are not using a visual component, you must explicitly open a `DataSet`. “Open” propagates up and “close” propagates down, so opening a `DataSet` implicitly opens a `Database`. A `Database` is never implicitly closed.

Ensuring that a query is updatable

When JBuilder executes a query, it attempts to make sure that the query is updatable and that it can be resolved back to the database. If JBuilder determines that the query is not updatable, it will try to modify the query to make it updatable, typically by adding columns to the `SELECT` clause.

If a query is found to not be updatable and JBuilder cannot make it updatable by changing the query, the resulting data set will be read-only.

To make any data set updatable, set the `updateMetaData` property to **NONE** and specify the data set’s table name and unique row identifier columns (some set of columns that can uniquely identify a row, such as columns of a primary or unique index). See [“Persisting query metadata” on page 52](#) for information on how to do this.

You can query a SQL view, but JBuilder will not indicate that the data was received from a SQL view as opposed to a SQL table, so there is a risk the data set will not be updatable. You can solve this problem by writing a custom resolver.

Using parameterized queries to obtain data from your database

A parameterized SQL statement contains variables, also known as parameters, the values of which can vary at run time. A parameterized query uses these variables to replace literal data values, such as those used in a WHERE clause for comparisons that appear in a SQL statement. These variables are called *parameters*. Ordinarily, parameters stand in for data values passed to the statement. You provide the values for the parameters before running the query. By providing different sets of values and running the query for each set, you cause one query to return different data sets.

An understanding of how data is provided to a `DataSet` is essential to further understanding of parameterized queries, so read [Chapter 2, “Understanding JBuilder database applications”](#) and [“Querying a database” on page 46](#) if you have not already done so. This topic is specific to parameterized queries.

In addition to the instructions provided in [“Parameterizing a query” on page 53](#), the following parameterized query topics are discussed:

- [“Using parameters” on page 57](#)
- [“Re-executing the parameterized query with new parameters” on page 59](#)
- [“Parameterized queries in master-detail relationships” on page 59](#)

Parameterizing a query

The following example shows how to provide data to an application using a `QueryDataSet` component. This example adds a `ParameterRow` with low and high values that can be changed at run time. When the values in the `ParameterRow` are changed, the table will automatically refresh its display to reflect only the records that meet the criteria specified with the parameters.

Note We strongly recommend that before starting the following steps you familiarize yourself with using the visual design tools by performing the tutorial in [Chapter 16, “Tutorial: Importing and exporting data from a text file.”](#)

A completed version of the application created with the following steps is available in the sample project `ParameterizedQuery.jpx`, located in the `/samples/DataExpress/ParameterizedQuery` directory of your JBuilder installation.

Creating the application

To create the application,

- 1 Choose `File|Close All`.
- 2 Choose `File|New` and double-click the Application icon.
- 3 Accept all defaults to create a new application.
- 4 Select the Design tab to activate the UI designer.
- 5 Click the `Database` component on the DataExpress page of the component palette, then click anywhere in the UI designer to add the component to the application.



Open the Connection property editor for the `Database` component by clicking the ellipsis (...) button in the `connection` property value in the Inspector.

- Set the connection properties to the JDataStore sample EMPLOYEE table, as follows:

Property name	Value
Driver	com.borland.datastore.jdbc.DataStoreDriver
URL	Browse to <jbuilder>/samples/JDataStore/datastores/employee.jds in the local URL field.
Username	Enter your name (the default is "SYSDBA")
Password	Enter your password (the default is "masterkey")

The Connection dialog includes a Test Connection button. Click this button to check that the connection properties have been correctly set. Results of the connection attempt are displayed beside the button. When the connection is successful, click OK.

If you want to see the code that was generated, click on the Source tab and look for the `ConnectionDescriptor` code. Click the Design tab to continue.

For more information on connecting to databases, see [Chapter 4, "Connecting to a database."](#)

Adding a Parameter Row

Next, you will add a `ParameterRow` with two columns: `low_no` and `high_no`. After you bind the `ParameterRow` to a `QueryDataSet`, you can use `JdbTextField` components to change the value in the `ParameterRow` so that the query can be refreshed using these new values.

- Add a `ParameterRow` component to the application from the DataExpress page.
- Click the expand icon to the left of `parameterRow1` in the component tree to display the columns contained in the `ParameterRow`.
- Select <new column>, and set the following properties for the new column in the Inspector:

Property name	Value
columnName	low_no
dataType	INT
default	15

To see the code generated by the designer for this step, click the Source tab and look at the `jbInit()` method. Click the Design tab to continue.

- Select <new column> again to add the second column to the `ParameterRow`, and set the following properties for it:

Property name	Value
columnName	high_no
dataType	INT
default	50

Adding a QueryDataSet

- Add a `QueryDataSet` component from the DataExpress page to the application.
- Click the ellipsis (...) button for the `query` property to open the Query property editor.

- 3 Set the `query` property for `queryDataSet1` as follows:

Property name	Value
Database	database1
SQL Statement	select emp_no, first_name, last_name from employee where emp_no >= :low_no and emp_no <= :high_no

- 4 Click the Parameters tab in the Query property editor.
- 5 Select `parameterRow1` in the drop-down list box to bind the data set to the `ParameterRow`.
- 6 Select the Query tab, and click the Test Query button to ensure that the query is runnable. When the area beneath the button indicates *Success*, click OK to close the dialog.

The following code for the `queryDescriptor` is added to the `jbInit()` method:

```
queryDataSet1.setQuery(new com.borland.dx.sql.dataset.QueryDescriptor(database1,
    "select emp_no, first_name, last_name from employee where
    emp_no <= :low_no and emp_no >= :high_no",
    parameterRow1, true, Load.ALL));
```

- 7 Add a `DBDisposeMonitor` component from the More dbSwing tab. The `DBDisposeMonitor` will close the `JDataStore` when the window is closed.
- 8 Set the `dataAwareComponentContainer` property for the `DBDisposeMonitor` to this.

Add the UI components

The following instructions assume you have followed the beginning database tutorial and are already familiar with adding UI components to the designer.

To add the components for viewing and manipulating the data in your application,

- 1 Click the `TableScrollPane` component on the dbSwing page of the component palette, and drop it into the center of the panel in the UI designer.

Make sure its `constraints` property is set to `CENTER`.

- 2 Drop a `JdbTable` component from the dbSwing page into the center of `tableScrollPane1` component, and set its `dataSet` property to `queryDataSet1`.

You'll notice that the table in the designer displays live data.

- 3 Choose Run/Run Project to run the application and browse the data set.
- 4 Close the running application.

To add the components that make the parameterized query variable at run time,

- 1 Select the `JPanel` component on the Swing Containers page, and drop it into the component tree, directly on the icon to the left of `contentPane(BorderLayout)`.

This ensures that the `JPanel` (`jPanel1`) will be added to the main UI, rather than to `tableScrollPane1` which is currently occupying the entire UI panel.

- 2 Make sure its `constraints` property is set to `NORTH`.

If `tableScrollPane1` suddenly shrinks, check that its `constraints` property is still set to `CENTER`.

- 3 Select `jPanel1` and set its `preferredSize` property to `200,100`.

This will make it big enough to contain the rest of the components for the UI.

- 4 Drop a `JdbTextField` component from the dbSwing page into `jPanel1`.

This component holds the minimum value.

- 5 Notice that `jdbTextField1` is placed in the center of the panel at the top.

This is because the default layout for a `JPanel` component is `FlowLayout`. If you try to drag the component to a different location, it won't stay there, but will return to its initial location.

To take control of the placement of the UI components in this panel, change the `layout` property for `jPanel1` to 'null'. Then, drag `jdbTextField1` to the left side of the panel.

- 6 Set the `columns` property for `jdbTextField1` to 10 to give it a fixed width. Set its `text` property to 10 to match the default minimum parameter value you entered earlier.
- 7 Add a `JLabel` from the Swing page to `jPanel1`. This label will identify `jdbTextField1` as the minimum field.
- 8 Click on `jLabel1` in the UI designer and drag it to just above `jdbTextField1`.
- 9 Set the `text` property for `jLabel1` to Minimum value. Grab the middle black sizing nib on the right edge and expand the width of the label until all of the text is visible.
- 10 Add another `JdbTextField` and `JLabel` to `jPanel1` for the maximum value. Drag this pair of components to the right side of the panel.
- 11 Set the `columns` property for `jdbTextField2` to 10, and its `text` property to 50.
- 12 Set the `text` property for `jLabel2` to Maximum value, and expand its width to show all the text.
- 13 Align all four components.

Hold the control key down and click on both `jLabel1` and `jdbTextField1`. Right-click and choose Align Left so their left edges will be aligned. (When you are using null layout for prototyping a UI, you have alignment options available from the context menu.)

Left align `jLabel2` and `jdbTextField2`. Top align the two text fields, and top align the two labels.

- 14 Add a `JButton` from the Swing page to `jPanel1`. Put this button in the middle, midway between the two text fields. Set its `text` property to Update.

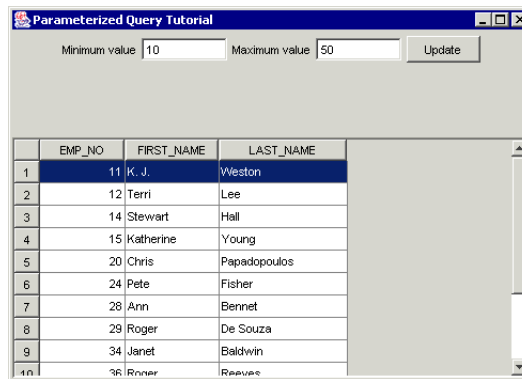
Clicking this button will update the results of the parameterized query with the values entered into the minimum and maximum value entry fields.

- 15 Select the Events tab of the Inspector, select the `actionPerformed` field, and double-click the value field to create an `actionPerformed()` event in the source code. The Source pane will display and the cursor will be located between the opening and closing braces for the new `actionPerformed()` event.

Add the following code so the event looks like this:

```
void jButton1_actionPerformed(ActionEvent e) {
    try {
        // change the values in the parameter row
        // and refresh the display
        parameterRow1.setInt("low_no",
            Integer.parseInt(jdbTextField1.getText()));
        parameterRow1.setInt("high_no",
            Integer.parseInt(jdbTextField2.getText()));
        queryDataSet1.refresh();
    }
    catch (Exception ex) {
        ex.printStackTrace();
    }
}
```


16 Save your work, and run the application. It should look like similar to this:



To test the example, enter a new value in the minimum value entry field, then click the Update button. The table displays only those values above the new minimum value. Enter a new value in the maximum value entry field, then click the Update button. The table displays only those values below the new maximum value.

To save changes back to the data source, you will need to add a `QueryResolver`. See [“Saving changes from a QueryDataSet” on page 78](#) to learn how to add a button with resolving code, or add a `JdbNavToolBar` component to the content pane and use its Save Changes button as a default query resolver.

Parameterized queries: Hints & tips

This set of topics includes tips to help you

- Determine how to use named parameters and parameter markers
- Re-execute the query with new parameters
- Use a parameterized query in a master-detail relationship

Using parameters

To assign parameter values in a parameterized query, you must first create a `ParameterRow` and add named columns that will hold the values to be passed to the query.

Any `ReadWriteRow`, such as `ParameterRow`, `DataSet`, and `DataRow` may be used as query or procedure parameters. In a `ParameterRow`, columns can simply be set up with the `addColumn`s and `setColumn`s methods. `DataSet` and `DataRow` should only be used if they already contain the columns with the wanted data.

The `Row` classes are used extensively in the `DataExpress` APIs. The `ReadRow` and `ReadWriteRow` are used much like interfaces that indicate the usage intent. By using a class hierarchy, implementation is shared, and there is a slight performance advantage over using interfaces.

The following text illustrates the class hierarchy associated with the `DataSet` methods:

```
java.lang.Object
+----com.borland.dx.dataset.ReadRow
+----com.borland.dx.dataset.ReadWriteRow
+----com.borland.dx.dataset.DataSet
+----com.borland.dx.dataset.StorageDataSet
+----com.borland.dx.sql.dataset.QueryDataSet
```

- `StorageDataSet` methods deal with data set structure
- `DataSet` methods handle navigation
- `ReadWriteRow` methods let you edit column values (that is, fields) in the current row
- `ReadRow` methods give read access to column values (that is, fields) in the current row
- `TableDataSet` and `QueryDataSet` inherit all these methods.

The `Row` classes provide access to column values by ordinal and column name. Specifying columns by name is a more robust and readable way to write your code. Accessing columns by name is not quite as quick as by ordinal, but it is still quite fast if the number of columns in your `DataSet` is less than twenty, due to some patented high-speed name/ordinal matching algorithms. It is also a good practice to use the same strings for all access to the same column. This saves memory and is easier to enter if there are many references to the same column.

The `ParameterRow` is passed in the `QueryDescriptor`. The `query` property editor allows you to select a parameter row. Editing of `ParameterRow`, such as adding a column and changing its properties, can be done in the Inspector or in code.

For example, you create a `ParameterRow` with two fields, `low_no` and `high_no`. You can refer to `low_no` and `high_no` in your parameterized query, and compare them to any field in the table. See the examples below for how to use these values in different ways.

In JBuilder, parameterized queries can be run with named parameters, with parameter markers, or with a master-detail relationship. The following sections give a brief explanation of each.

- With named parameters:

When the parameter markers in the query are specified with a colon followed by an alphanumeric name, parameter name matching will be done. The column in the `ParameterRow` that has the same name as a parameter marker will be used to set the parameter value. For example, in the following SQL statement, values to select are passed as named parameters:

```
SELECT * FROM employee where emp_no > :low_no and emp_no < :high_no
```

In this SQL statement, `:low_no` and `:high_no` are parameter markers that are placeholders for actual values supplied to the statement at run time by your application. The value in this field may come from a visual component or be generated programmatically. In design time, the column's default value will be used. When parameters are assigned a name, they can be passed to the query in any order. JBuilder will bind the parameters to the data set in the proper order at run time.

In [“Parameterizing a query” on page 53](#), two columns are added to the `ParameterRow` to hold minimum and maximum values. The query descriptor specifies that the query should return only values greater than the minimum value and less than the maximum value.

- With ? JDBC parameter markers:

When the simple question mark JDBC parameter markers are used, parameter value settings are ordered strictly from left to right.

For example, in the following SQL statement, values to select are passed as ? JDBC parameters markers:

```
SELECT * FROM employee WHERE emp_no > ?
```

In this SQL statement, the “?” value is a placeholder for an actual value supplied to the statement at run time by your application. The value in this field may come from a visual component or be generated programmatically. When a ? JDBC parameter marker is used, values are passed to the query in a strictly left to right order.

JBuilder will bind the parameters to the source of the values (a `ReadWriteRow`) in this order at run time. Binding parameters means allocating resources for the statement and its parameters both locally and on the server in order to improve performance when a query is executed.

- With a master-detail relationship:

Master and detail data sets have at least one field in common, by definition. This field is used as a parameterized query. For more detail on supplying parameters in this way, see [“Parameterized queries in master-detail relationships” on page 59](#).

Re-executing the parameterized query with new parameters

To re-execute the query with new parameters, set new values in the `ParameterRow` and then call `QueryDataSet.refresh()` to cause the query to be executed again with new parameter values. For example, to use a UI component to set the value of a parameter, you can use a SQL statement such as:

```
SELECT * FROM phonelist WHERE lastname LIKE :searchname
```

In this example, the `:searchname` parameter's value could be supplied from a UI component. To do this, your code would have to:

- 1 Obtain the value from the component each time it changes
- 2 Place it into the `ParameterRow` object
- 3 Supply that object to the `QueryDataSet`
- 4 Call `refresh()` on the `QueryDataSet`

See [“Parameterizing a query” on page 53](#) for an example of how to do this with JBuilder sample files.

If the values you want to assign to the query parameter exist in a column of a data set, you can use that data set as your `ReadWriteRow` in the `QueryDescriptor`, navigate through the data set, and rerun the query for each value.

Parameterized queries in master-detail relationships

In a master-detail relationship with `DelayedDetailFetch` set to `true` (to fetch details when needed), you can specify a SQL statement such as:

```
SELECT * FROM employee WHERE country = :job_country
```

In this example, `:job_country` would be the field that this detail data set is using to link to a master data set. You can specify as many parameters and master link fields as is necessary. In a master-detail relationship, the parameter must always be assigned a name that matches the name of the column. For more information about master-detail relationships and the `DelayedDetailFetch` parameter, see [Chapter 9, “Establishing a master-detail relationship.”](#)

In a master-detail descriptor, binding is done implicitly. Implicit binding means that the data values are not actually supplied by the programmer, they are retrieved from the master row and implicitly bound when the detail query is executed. Binding parameters means allocating resources for the statement and its parameters both locally and on the server in order to improve performance when a query is executed.

If the values you want to assign to the query parameter exist in a column of a data set (the master data set), you can use that data set as your `ReadWriteRow` in the `QueryDescriptor`, navigate through the data set, and rerun the query for each value to display in the detail data set.

Using stored procedures

With a stored procedure, one or more SQL statements are encapsulated in a single location on your server and can be run as a batch. `ProcedureDataSet` components enable you to access, or provide, data from your database with existing stored procedures, invoking them with either JDBC procedure escape sequences or server-specific syntax for procedure calls. To run a stored procedure against a SQL table where the output is a set of rows, you need the following components. You can provide this information programmatically, or by using JBuilder design tools.

- The `Database` component encapsulates a database connection through JDBC to the SQL server and also provides lightweight transaction support.
- The `ProcedureDataSet` component provides the functionality to run the stored procedure (with or without parameters) against the SQL database and stores the results from the execution of the stored procedure.
- The `ProcedureDescriptor` object stores the stored procedure properties, including the database to be queried, the stored procedures, escape sequences, or procedure calls to execute, and any optional stored procedure parameters.

When providing data from JDBC data sources, the `ProcedureDataSet` has built-in functionality to fetch data from a stored procedure that returns a cursor to a result set. The following properties of the `ProcedureDescriptor` object affect the execution of stored procedures:

Property	Purpose
database	Specifies what <code>Database</code> connection object to run the procedure against.
procedure	A Java String representation of a stored procedure escape sequence or SQL statement that causes a stored procedure to be executed.
parameters	An optional <code>ReadWriteRow</code> from which to fill in parameters. These values can be acquired from any <code>DataSet</code> or <code>ReadWriteRow</code> .

Property	Purpose
<code>executeOnOpen</code>	Causes the <code>ProcedureDataSet</code> to execute the procedure when it is first opened. This is useful for presenting live data at design time. You may also want this enabled at run time. The default value is true .
<code>loadOption</code>	An optional integer value that defines the method of loading data into the data set. Options are: <ol style="list-style-type: none"> 1 Load All Rows: load all data up front. 2 Load Rows Asynchronously: causes the fetching of <code>DataSet</code> rows to be performed on a separate thread. This allows the <code>DataSet</code> data to be accessed and displayed as the <code>QueryDataSet</code> is fetching rows from the database connection. 3 Load As Needed: load the rows as they are needed. 4 Load 1 Row At A Time: load as needed and replace the previous row with the current. Useful for high-volume batch-processing applications.

A `ProcedureDataSet` can be used to run stored procedures with and without parameters. A stored procedure with parameters can acquire the values for its parameters from any `DataSet` or `ParameterRow`. The section [“Using parameters with Oracle PL/SQL stored procedures” on page 64](#) provides an example.

Use Database Pilot to browse and edit database server-specific schema objects, including tables, fields, stored procedure definitions, triggers, and indexes. For more information on Database Pilot, choose Tools|Database Pilot and refer to its online help.

The following topics related to stored procedure components are covered:

- [“Escape sequences, SQL statements, and server-specific procedure calls” on page 62](#)
- [“Using InterBase stored procedures” on page 63](#)
- [“Using parameters with Oracle PL/SQL stored procedures” on page 64](#)
- [“Using Sybase stored procedures” on page 65](#)

Stored procedures: hints & tips

This section contains tips to help you understand the options for using a stored procedure.

Note For information on using stored procedures with `JDataStore`, see “Stored Procedures and UDFs” in the *JDataStore Developer’s Guide*.

Escape sequences, SQL statements, and server-specific procedure calls

When entering information in the Stored Procedure Escape or SQL Statement field in the `procedure` property editor, or in code, you have three options for the type of statement to enter. These are

- Select an existing procedure.

To browse the database for an existing procedure, click Browse Procedures in the `procedure` property editor. A list of available procedure names for the database you are connected to is displayed. If the server is InterBase and you select a procedure that does not return data, you receive a notice to that effect. If you select a procedure that does return data, JBuilder attempts to generate the correct escape syntax for that procedure call. However, you may need to edit the automatically-generated statement to correspond correctly to your server’s syntax. For other

databases, only the procedure name is inserted from the Select Procedure dialog box.

If the procedure is expecting parameters, you have to match these with the column names of the parameters.

- Enter a JDBC procedure escape sequence.

To enter a JDBC procedure escape sequence, use the following formatting:

```
{call PROCEDURENAME (?,?,?,...)} for procedures
```

```
{?= call FUNCTIONNAME (?,?,?,...)} for functions
```

- Enter server-specific syntax for procedure calls.

When a server allows a separate syntax for procedure calls, you can enter that syntax instead of an existing stored procedure or JDBC procedure escape sequence. For example, server-specific syntax may look like this:

```
execute procedure PROCEDURENAME ?,?,?
```

In both of the last two examples, the parameter markers, or question marks, may be replaced with named parameters of the form :ParameterName. For an example using named parameters, see [“Using parameters with Oracle PL/SQL stored procedures” on page 64](#). For an example using InterBase stored procedures, see [“Using InterBase stored procedures” on page 63](#).

Using vendor-specific stored procedures

This section contains information to help you use stored procedures with specific database vendors. Information is provided to help you use the following types of stored procedures:

- JDataStore stored procedures and user-defined functions
- InterBase stored procedures
- Oracle PL/SQL stored procedures
- Sybase stored procedures

Using JDataStore stored procedures and user-defined functions

JDataStore 6 supports the use of Java-based stored procedures and user-defined functions (UDFs). Stored procedures and UDFs must be added to the CLASSPATH of the JDataStore server process. Stored procedures and UDFs for JDataStore must be written in Java. UDFs are user-defined functions that are designed to be used in subexpressions of a SQL statement.

For more information, including usage instructions and examples, see “UDFs and Stored Procedures” in the *JDataStore Developer’s Guide*.

Using InterBase stored procedures

In InterBase, the SELECT procedures may be used to generate a `DataSet`. In the InterBase sample database, `employee.gdb`, the stored procedure `ORG_CHART` is such a procedure. To call this procedure from JBuilder, enter the following syntax in the Stored Procedure Escape or SQL Statement field in the `procedure` property editor, or in code:

```
select * from ORG_CHART
```

For a look at more complicated InterBase stored procedures, use Database Pilot to browse procedures on this server. ORG_CHART is an interesting example. It returns a result set that combines data from several tables. ORG_CHART is written in InterBase's procedure and trigger language, which includes SQL data manipulation statements plus control structures and exception handling.

The output parameters of ORG_CHART turn into columns of the produced `DataSet`.

See the InterBase Server documentation for more information on writing InterBase stored procedures.

Using parameters with Oracle PL/SQL stored procedures

Currently, a `ProcedureDataSet` can only be populated with Oracle PL/SQL stored procedures if you are using Oracle's type-2 or type-4 JDBC drivers. The stored procedure that is called must be a function with a return type of `CURSOR REF`.

Follow this general outline for using Oracle stored procedures in JBuilder:

1 Define the function using PL/SQL.

The following is an example of a function description defined in PL/SQL that has a return type of `CURSOR REF`. This example assumes that a table named `MyTable1` exists.

```
create or replace function MyFct1(INP VARCHAR2) RETURN rcMyTable1 as
  type rcMyTable1 is ref cursor return MyTable1%ROWTYPE;
  rc rcMyTable1;
begin
  open rc for select * from MyTable1;
  return rc;
end;
```

2 Set up a `ParameterRow` to pass to the `ProcedureDescriptor`.

The input parameter `INP` should be specified in the `ParameterRow`, but the special return value of a `CURSOR REF` should not. JBuilder will use the output of the return value to fill the `ProcedureDataSet` with data. An example for doing this with a `ParameterRow` follows.

```
ParameterRow row = new ParameterRow();

row.addColumn( "INP", Variant.STRING, ParameterType.IN);

row.setString("INP", "Input Value");

String proc = "{?=call MyFct1(?)}";
```

3 Select the Frame file in the project pane, then select the Design tab.

4 Place a `ProcedureDataSet` component from the DataExpress page of the component palette on the design surface.

5 Select the `procedure` property to bring up the `ProcedureDescriptor` dialog box.

6 Select `database1` from the Database drop-down list.

7 Enter the following escape syntax in the Stored Procedure Escape or SQL Statement field, or in code:

```
{?=call MyFct1(?)}
```

8 Select the Parameters tab of the dialog box. Select the `ParameterRow` just defined as `row`.

See your Oracle server documentation for information on the Oracle PL/SQL language.

Using Sybase stored procedures

Stored procedures created on Sybase servers are created in a “chained” transaction mode. In order to call Sybase stored procedures as part of a `ProcedureResolver`, the procedures must be modified to run in an unchained transaction mode. To do this, use the Sybase stored system procedure `sp_procmode` to change the transaction mode to either “anymode” or “unchained.” For more details, see the Sybase documentation.

Sample application with database-server specific stored procedures

In the `<jbuilder>/samples/DataExpress/ServerSpecificProcedures` directory, you can browse a sample application with sample stored procedure code for `JDataStore`, Sybase, InterBase, and Oracle databases.

Writing a custom data provider

JBuilder makes it easy to write a custom provider for your data when you are accessing data from a custom data source, such as SAP, BAAN, IMS, OS/390, CICS, VSAM, DB2, etc.

The retrieval and update of data from a data source, such as an Oracle or Sybase server, is isolated to two key interfaces: providers and resolvers. *Providers* populate a data set from a data source. *Resolvers* save changes back to a data source. By cleanly isolating the retrieval and updating of data to two interfaces, it is easy to create new provider/resolver components for new data sources. JBuilder currently provides implementations for standard JDBC drivers that provide access to popular databases such as support for Oracle, Sybase, Informix, InterBase, DB2, MS SQL Server, Paradox, dBASE, FoxPro, Access, and other popular databases. These include,

- `OracleProcedureProvider`
- `ProcedureProvider`
- `ProcedureResolver`
- `QueryProvider`
- `QueryResolver`

You can create custom provider/resolver component implementations for EJB, application servers, SAP, BAAN, IMS, CICS, etc.

An example project with a custom provider and resolver is located in the `/samples/DataExpress/CustomProviderResolver` directory of your JBuilder installation. The sample file `TestFrame.java` is an application with a frame that contains a `JdbTable` and a `JdbNavToolBar`. Both visual components are connected to a `TableDataSet` component where data is provided from a custom `Provider` (defined in the file `ProviderBean.java`), and data is saved with a custom `Resolver` (defined in the file `ResolverBean.java`). This sample application reads from and saves changes to the text file `data.txt`, a simple non-delimited text file. The structure of `data.txt` is described in the interface file `DataLayout.java`.

This topic discusses custom data providers, and how they can be used as providers for a `TableDataSet` and any `DataSet` derived from `TableDataSet`. The main method to implement is `provideData(com.borland.dx.dataset.StorageDataSet dataSet, boolean toOpen)`. This method accesses relevant metadata and loads the actual data into the data set.

Obtaining metadata

Metadata is information *about* the data. Examples of metadata are column name, table name, whether the column is part of the unique row id or not, whether it is searchable, its precision, scale, and so on. This information is typically obtained from the data source. The metadata is then stored in properties of `Column` components for each column in the `StorageDataSet`, and in the `StorageDataSet` itself.

When you obtain data from a data source, and store it in one of the subclasses of `StorageDataSet`, you typically obtain not only rows of data from the data source, but also metadata. For example, the first time that you ask a `QueryDataSet` to perform a query, by default it runs two queries: one for metadata discovery and the second for fetching rows of data that your application displays and manipulates. Subsequent queries performed by that instance of `QueryDataSet` only do row data fetching. After discovering the metadata, the `QueryDataSet` component then creates `Column` objects automatically as needed at run time. One `Column` is created for every query result column that is not already in the `QueryDataSet`. Each `Column` then gets some of its properties from the metadata, such as `columnName`, `tableName`, `rowId`, `searchable`, `precision`, `scale`, and so on.

When you are implementing the abstract `provideData()` method from the `Provider` class, the columns from the data provided may need to be added to your `DataSet`. This can be done by calling the `ProviderHelp.initData()` method from inside your `provideData()` implementation. Your provider should build an array of `Columns` to pass to the `ProviderHelp.initData()` method. The following is a list of `Column` properties that a `Provider` should consider initializing:

- `columnName`
- `dataType`

and optionally,

- `sqlType`
- `precision` (used by `DataSet`)
- `scale` (used by `DataSet`)
- `rowId`
- `searchable`
- `tableName`
- `schemaName`
- `serverColumnName`

The optional properties are useful when saving changes back to a data source. The `precision` and `scale` properties are also used by `DataSet` components for constraint and display purposes.

Invoking initData

The arguments to the `ProviderHelp.initData(com.borland.dx.dataset.StorageDataSet dataSet, com.borland.dx.dataset.Column[], boolean, boolean, boolean)` method are explained in the following text.

- `dataSet` is the `StorageDataSet` we are providing to
- `metaDataColumns` is the `Column` array created with the proper properties that do not need to be added/merged into the `Columns` that already exist in `DataSet`
- `updateColumns` specifies whether to merge columns into existing persistent columns that have the same `columnName` property setting
- `keepExistingColumns` specifies whether to keep any non-persistent columns

If `keepExistingColumns` is `true`, non-persistent columns are also retained. Several column properties in the columns array are merged with existing columns in the `StorageDataSet` that have the same `name` property setting. If the number, type, and position of columns is different, this method may close the associated `StorageDataSet`.

The `metaDataUpdate` property on `StorageDataSet` is investigated when `ProviderHelp.initData` is called. This property controls which `Column` properties override properties in any persistent columns that are present in the `TableDataSet` before `ProviderHelp.initData` is called. Valid values for this property are defined in the `MetaDataUpdate` interface.

Obtaining actual data

Certain key `DataSet` methods cannot be used when the `Provider.provideData` method is called to open a `DataSet`, while the `DataSet` is in the process of being opened, including the `StorageDataSet.insertRow()` method.

In order to load the data, use the `StorageDataSet.startLoading` method. This method returns an array of `Variant` objects for all columns in a `DataSet`. You set the value in the array (the ordinal values of the columns are returned by the `ProviderHelp.initData` method), then load each row by calling the `StorageDataSet.loadRow()` method, and finish by calling the `StorageDataSet.endLoading()` method.

Tips on designing a custom data provider

A well designed provider recognizes the `maxRows` and `maxDesignRows` properties on `StorageDataSet`. The values for these properties are,

Value	Description
0	provide metadata information only
-1	provide all data
n	provide maximum of n rows

To determine if the `provideData()` method was called while in design mode, call `java.beans.Beans.isDesignTime()`.

Understanding the `provideData()` method in master-detail data sets

The `Provider.provideData()` method is called

- when the `StorageDataSet` is initially opened (`toOpen` is `true`)
- when `StorageDataSet.refresh()` is called
- when a detail data set with the `fetchAsNeeded` property set to `true` needs to be loaded

When a detail data set with the `fetchAsNeeded` property set to `true` needs to be loaded, the provider ignores `provideData` during the opening of the data, or just provides the metadata. The provider also uses the values of the `masterLink` fields to provide the rows for a specific detail data set.

Working with columns

A `Column` is the collection of one type of information (for example, a collection of phone numbers or job titles). A collection of `Column` components are managed by a `StorageDataSet`.

A `Column` object can be created explicitly in your code, or generated automatically when you instantiate the `StorageDataSet` subclass, for example, by a `QueryDataSet` when a query is executed. Each `Column` contains properties that describe or manage that column of data. Some of the properties in `Column` hold *metadata* (defined below) that is typically obtained from the data source. Other `Column` properties are used to control its appearance and editing in data-aware components.

Note Abstract or superclass class names are often used to refer generally to all their subclasses. For example, a reference to a `StorageDataSet` object implies any one (or all, depending on its usage) of its subclasses `QueryDataSet`, `TableDataSet`, `ProcedureDataSet`, and `DataSetView`.

Understanding Column properties and metadata

Most properties on a `Column` can be changed without closing and re-opening a `DataSet`. However, the following properties cannot be set unless the `DataSet` is closed:

- `columnName`
- `dataType`
- `calcType`
- `pickList`
- `preferredOrdinal`

The UI designer will do live updates for `Column` display-oriented properties such as `color`, `width`, and `caption`. For more information on obtaining metadata, see [“Obtaining metadata” on page 66](#). For more discussion on obtaining actual data, see [“Obtaining actual data” on page 67](#).

Non-metadata Column properties

Columns have additional properties that are not obtained from metadata that you may want to set, for example, `caption`, `editMask`, `displayMask`, `background` and `foreground` colors, and `alignment`. These types of properties are typically intended to control the

default appearance of this data item in data-aware components, or to control how it can be edited by the user. The properties you set in an application are usually of the non-metadata type.

Viewing column information in the column designer

One way to view column properties information is by using the column designer. The column designer displays information for selected properties, such as the data type for the column, in a navigable table. Changing, or setting, a property in the column designer makes a column persistent. The column properties can be modified in the column designer or in the Inspector. You can change which properties display in the column designer by clicking the Properties button.

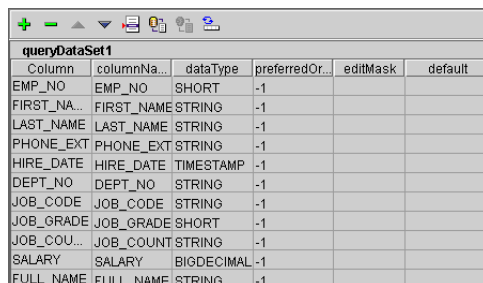
To display the column designer,

- 1 Open any project that includes a `DataSet` object.

In this example, open `/samples/DataExpress/QueryProvider/QueryProvider.jpx` from your JBuilder installation.

- 2 Double-click the file `Frame1.java` in the project pane and click the Design tab from the bottom of the right pane of the IDE.
- 3 Double-click the `queryDataSet1` object in the component tree.

This displays the column designer for the data set. The column designer looks like this for the EMPLOYEE sample table:



Column	columnName	dataType	preferredOrder	editMask	default
EMP_NO	EMP_NO	SHORT	-1		
FIRST_NAME	FIRST_NAME	STRING	-1		
LAST_NAME	LAST_NAME	STRING	-1		
PHONE_EXT	PHONE_EXT	STRING	-1		
HIRE_DATE	HIRE_DATE	TIMESTAMP	-1		
DEPT_NO	DEPT_NO	STRING	-1		
JOB_CODE	JOB_CODE	STRING	-1		
JOB_GRADE	JOB_GRADE	SHORT	-1		
JOB_COUNT	JOB_COUNT	STRING	-1		
SALARY	SALARY	BIGDECIMAL	-1		
FULL_NAME	FULL_NAME	STRING	-1		

To set a property for a column, select that `Column` and enter or select a new value for that property. The Inspector updates to reflect the properties (and events) of the selected column. For example,



- 1 Click the Properties button to open the Properties To Display dialog box.
- 2 Check the `min` property to display in the column designer, and click OK.
- 3 Scroll to the `min` column, and enter today's date for the HIRE_DATE field, using the following date format:

YYYY-MM-DD hh:mm:ss.nnnnnnnnn

where YYYY-MM-DD is the year, month, and day, and hh:mm:ss.nnnnnnnnn is the hour, minutes, seconds, and (optionally) nanoseconds.

- 4 Press *Enter* to change the value.

To close the column designer, select any UI component in the content pane. In other words, the only way to close one designer is to open a different one.

See the topic [“Ensuring data persistence” on page 153](#) for more information on using the column designer.

Generate RowIterator Class button

The RowIterator Generator in the column designer can be used to create a new `RowIterator` class or update an existing `RowIterator` class for a `DataSet`. It looks at the `columnName` property of all the `Columns` in the `DataSet`, and generates `get` and `set` methods for each column.

Selecting the RowIterator Generator button opens a dialog that provides lightweight (low memory usage and fast binding) iteration capabilities to ensure static type-safe access to columns.

The options in the RowIterator dialog have the following purposes:

Table 7.1 RowIterator Generator dialog

Option	Description
Extend RowIterator	If set, the generated class will extend <code>RowIterator</code> . This will surface all methods in <code>RowIterator</code> . If this is false, a new class with a <code>RowIterator</code> member will be created, and which is delegated for all operations. The advantage of not extending <code>RowIterator</code> is that your iterator class can control what gets exposed. The advantage of extending <code>RowIterator</code> is that less code needs to be generated due to the fact that binder and navigation methods are inherited and do not need to be delegated to.
Remove Underscore; Capitalize Next Letter	This affects how the <code>get</code> and <code>set</code> method names are generated from the <code>columnName</code> property of the <code>Column</code> . If this option is set, underscores are removed and the character following the underscore is capitalized.
Generate binder methods	Generates delegator methods to call the embedded <code>RowIterator</code> bind methods.
Generate navigation methods	Generates delegator methods to call the embedded <code>RowIterator</code> navigation methods.

For more information on `RowIterators`, see the *DataExpress Component Library Reference*.

Using the column designer to persist metadata

Clicking the Persist All Metadata button in the column designer will persist all the metadata that is needed to open a `QueryDataSet` at run time.

The source will be changed with these settings:

- The query of the `QueryDataSet` will be changed to include row identifier columns.
- The `metaDataUpdate` property of the `QueryDataSet` will be set to `NONE`.
- The `tableName`, `schemaName`, and `resolveOrder` properties on the `QueryDataSet` will be set, if needed.
- All columns will be persisted, with miscellaneous properties set. These properties are `precision`, `scale`, `rowId`, `searchable`, `tableName`, `schemaName`, `hidden`, `serverColumnName`, and `sqlType`.

JBuilder fetches metadata automatically. Because some JDBC drivers are slow at responding to metadata inquiries, you might want to persist metadata and tell DataExpress not to fetch it. With JBuilder setting this up at design time, and generating the necessary code for run time, performance will be improved.

See also

- [“Persisting query metadata” on page 52](#)

Making metadata dynamic using the column designer

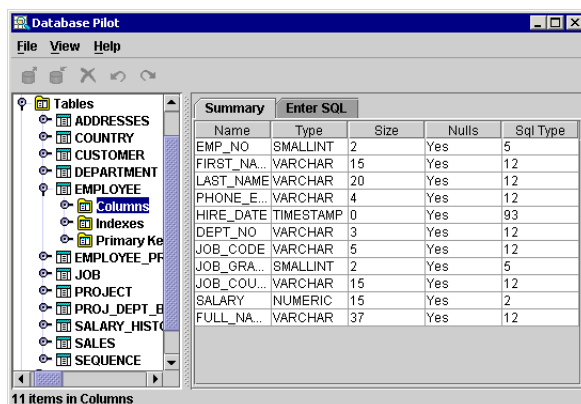
Warning Pressing the Make All Metadata Dynamic button will REMOVE CODE from the source file. It will remove all the code from the property settings mentioned in the previous topic, as well as any settings of the metadata-related properties named above. However, other properties, like `editMask` will not be touched.

Note To update a query after the table may have changed on the server, you must first make the metadata dynamic, then persist it, in order to use new indices created on the database table.

Viewing column information in the Database Pilot

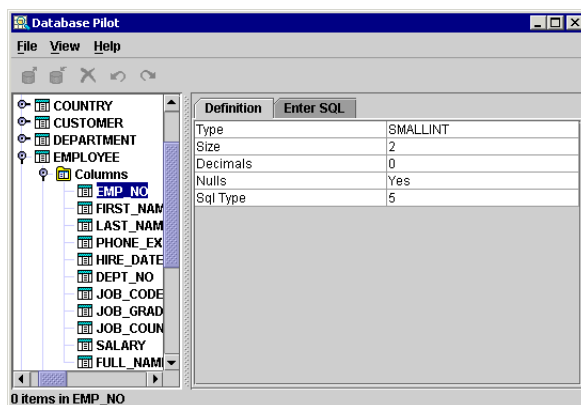
The Database Pilot is an all-Java, hierarchical database browser that also allows you to edit data. It presents JDBC-based meta-database information in a two-paneled window. The left pane contains a tree that hierarchically displays a set of databases and its associated tables, views, stored procedures, and metadata. The right pane is a multi-page display of descriptive information for each node of the tree.

To display the Database Pilot, choose Tools|Database Pilot from the JBuilder menu.



When a database URL is opened, you can expand the tree to display child objects. Columns are child objects of a particular database table. As in the figure above, when the Column object is selected for a table, the Summary page in the right pane lists the columns, their data type, size, and other information.

Select a column in the left pane to see just the information for that field, as in the figure below.



Refer to the Database Pilot online help for additional information.

Optimizing a query

This section contains information about how to work with columns to improve query performance.

Setting column properties

You can set `Column` properties through the JBuilder visual design tools or in code manually. Any column that you define or modify through the visual design tools will be persistent.

Setting Column properties using JBuilder's visual design tools

The Inspector allows you to work with `Column` properties. To set `Column` properties:

- 1 Open (or create) a project that contains a `StorageDataSet` that you want to work with. If you are creating a new project, you could follow the instructions in [“Querying a database” on page 46](#).
- 2 Open the UI designer by double-clicking the `Frame` container object in the project pane, and then clicking the Design tab.
- 3 In the content pane, select the `StorageDataSet` component.
- 4 Click the expand icon beside the `StorageDataSet` to display its columns.
- 5 Select the `Column` you want to work with. The Inspector displays the column's properties and events. Set the properties you want.

Setting properties in code

To set properties manually in your source code on one or more columns in a `StorageDataSet`:

- 1 Provide data to the `StorageDataSet`.
For example, run a query using a `QueryDataSet` component. See [“Querying a database” on page 46](#) for an example.
- 2 Obtain an array of references to the existing `Column` objects in the `StorageDataSet` by calling the `getColumn(java.lang.String)` method of the `ReadRow`.
- 3 Identify which column or columns in the array you want to work with by reading their properties, for example using the `getColumnName()` property of the `Column` component.
- 4 Set the properties on the appropriate columns as needed.

Note If you want the property settings to remain in force past the next time that data is provided, you must set the column's `persist` property to **true**. This is described in the following section.

Persistent columns

A persistent column is a `Column` object which was already part of a `StorageDataSet`, and whose `persist` property was set to **true** before data was provided. If the `persist` property is set after data is provided, you must perform another `setQuery` command with a new `queryDescriptor` for the application to recognize that the columns are persistent. A persistent `Column` allows you to keep `Column` property settings across a data-provide operation. A persistent column does not cause the data in that column of the data rows to freeze across data provide operations.

Normally, a `StorageDataSet` automatically creates new `Column` objects for every column found in the data provided by the data source. It discards any `Column` objects that were explicitly added previously, or automatically created for a previous batch of data. This

discarding of previous `Column` objects could cause you to lose property settings on the old `Column` which you might want to retain.

To avoid this, mark a `Column` as persistent by setting its `persist` property to `true`. When a column is persistent, the `Column` is not discarded when new data is provided to the `StorageDataSet`. Instead, the existing `Column` object is used again to control the same column in the newly-provided data. The column matching is done by column name.

Any column that you define or modify through the visual design tools will be persistent. Persistent columns are discussed more thoroughly in “Ensuring data persistence” on page 153. You can create `Column` objects explicitly and attach them to a `StorageDataSet`, using either `addColumn()` to add a single `Column`, or `setColumns()` to add several new columns at one time.

When using `addColumn`, you must set the `Column` to persistent prior to obtaining data from the data source or you will lose all of the column’s property settings during the provide. The `persist` property is set automatically with the `setColumns` method.

Note The UI designer calls the `StorageDataSet.setColumns()` method when working with columns. If you want to load and modify your application in the UI designer, use the `setColumns` method so the columns are recognized at design time. At run time, there is no difference between `setColumns` and `addColumn`.

Combining live metadata with persistent columns

During the providing phase, a `StorageDataSet` first obtains metadata from the data source, if possible. This metadata is used to update any existing matching persistent columns, and to create other columns that might be needed. The `metaDataUpdate` property of the `StorageDataSet` class controls the extent of the updating of metadata on persistent columns.

Removing persistent columns

This section describes how to undo column persistence so that a modified query no longer returns the (unwanted) columns in a `StorageDataSet`.

When you have a `QueryDataSet` or `TableDataSet` with persistent columns, you declare that these columns will exist in the resulting `DataSet` whether or not they still exist in the corresponding data source. But what happens if you no longer want these persistent columns?

When you alter the query string of a `QueryDataSet`, your old persistent columns are not lost. Instead, the new columns obtained from running the query are appended to your list of columns. You may make any of these new columns persistent by setting any of their properties.

Note When you expand a `StorageDataSet` by clicking its expand icon in the content pane, the list of columns does not change automatically when you change the query string. To refresh the columns list based on the results of the modified query, double click the `QueryDataSet` in the content pane. This executes the query again and appends any new columns found in the modified query.

To delete a persistent column you no longer need, select it in the content pane and press the `Delete` key, or select the column in the column designer and click the `Delete` button on the toolbar. This causes the following actions:

- The column is marked as non-persistent
- Any code that sets properties of this column is removed
- Any event handler logic you may have placed on this column is removed.

To verify that a deleted persistent column is no longer part of the `QueryDataSet`, double-click the data set in the content pane. This re-executes the query and displays all the columns in the resulting `QueryDataSet`.

Using persistent columns to add empty columns to a DataSet

On occasion you may want to add one or more extra columns to a `StorageDataSet`, columns that are not provided from the data source and that are not intended to be resolved back to the data source. For example, you might add extra columns under the following circumstances or in the following ways:

- Add an extra column for internal utility purposes. If you want to hide the column from displaying in data-aware components, set the `visible` property of the `Column` to `false`.
- Construct a new `DataSet` manually by adding the columns you want before computing the data stored in its rows.
- Construct a new `DataSet` to store data from a custom data source that isn't supported by JBuilder's providers and therefore doesn't provide metadata automatically.

In such cases, you can explicitly add a `Column` to the `DataSet`, before or after providing data. The `columnName` must be unique and cannot duplicate a name that already exists in the provided data. Additionally, if you will be providing data after adding the `Column`, be sure to mark the `Column` persistent so that the `Column` is not discarded when new data is provided.

To add a column manually in source code, follow the instructions in [“Persistent columns” on page 73](#).

To add a column manually using the JBuilder visual design tools,

- 1 Follow the first 3 steps in [“Setting Column properties using JBuilder's visual design tools” on page 73](#) to obtain the metadata into the columns listed in the content pane.

You can skip the steps for providing data if you want to add columns to an empty `DataSet`.

- 2 Select `<new column>`.

This option appears at the bottom of the list of columns.

- 3 In the Inspector, set the `columnName`, making sure that it is different from existing column names.

- 4 Set any other properties as needed for this new column.

JBuilder creates code for a new persistent `Column` object and attaches it to your `DataSet`. The new `Column` exists even before the data is provided. Because its name is dissimilar from any provided column names, this `Column` is not populated with data during the providing phase; all rows in this `Column` have `null` values.

Controlling column order in a DataSet

When a `StorageDataSet` is provided data, it performs the following actions:

- Deletes any non-persistent columns, moving the persistent columns to the left.
- Merges columns from the provided data with persistent columns. If a persistent column has the same name and data type as a provided column, it is considered to be the same column.
- Places the provided columns into the data set in the order specified in the query or procedure.

- Adds the remaining columns—those defined only in the application—in the order they are defined in the data set's `setColumns()` method.
- Tries to move every column whose `preferredOrdinal` property is set to its desired place. (If two columns have the same `preferredOrdinal`, this won't be possible.)

This means that,

- Columns that are defined in your application and that are not provided by the query or procedure will appear after columns that are provided.
- Setting properties on some columns (whether provided or defined in the application), but not others, will not change their order.
- You can change the position of any column by setting its `preferredOrdinal` property. Columns whose `preferredOrdinal` is not set retain their position relative to each other.

Saving changes back to your data source

After data has been retrieved from a data source, and you make changes to the data in the `StorageDataSet`, you will want to save the changes back to your data source. All recorded changes to a `DataSet` can be saved back to a data source such as a SQL server. This process is called *resolving*. Sophisticated built-in reconciliation technology deals with potential edit conflicts.

Between the time that the local subset of data is retrieved from a data source, and the time that you attempt to save updates back to the data source, various situations may arise that must be handled by the resolver logic. For example, when you attempt to save your changes, you may find that the same information on the server has been updated by another user. Should the resolver save the new information regardless? Should it display the updated server information and compare it with your updates? Should it discard your changes? Depending on your application, the need for resolution rules will vary.

The logic involved in resolving updates can be fairly complex. Errors can occur while saving changes, such as violations of server integrity constraints and resolution conflicts. A resolution conflict may occur, for example, when deleting a row that has already been deleted, or updating a row that has been updated by another user. JBuilder provides default handling of these errors by positioning the `DataSet` to the offending row (if it's not deleted) and displaying the error encountered with a message dialog.

When resolving changes back to the data source, these changes are normally batched in groups called *transactions*. The DataExpress mechanism uses a single transaction to save all inserts, updates, and deletions made to the `DataSet` back to the data source by default. To allow you greater control, JBuilder allows you to change the default transaction processing.

DataExpress also provides a generic resolver mechanism consisting of base classes and interfaces. You can extend these to provide custom resolver behavior when you need greater control over the resolution phase. This generic mechanism also allows you to create resolvers for non-JDBC data sources that typically do not support transaction processing.

The following topics discuss the options for resolving data:

- [“Saving changes from a QueryDataSet” on page 78](#) covers the basic resolver handling provided by DataExpress and its default transaction processing.
When a master-detail relationship has been established between two or more data sets, special resolving procedures are required. For more information, see [“Saving changes in a master-detail relationship” on page 100](#).
- [“Saving changes back to your data source with a stored procedure” on page 80](#) covers resolving changes made to a `ProcedureDataSet` back to its data source.
- [“Resolving data from multiple tables” on page 83](#) provides the necessary settings for resolving changes when a query involves more than one table.
- [“Using DataSets with RMI \(streamable data sets\)” on page 85](#) provides a way to stream the data of a `DataSet` by creating a Java Object (`DataSetData`) that contains data from a `DataSet`.
- [“Customizing the default resolver logic” on page 87](#) describes how to set custom resolution rules using the `QueryResolver` component and resolver events.
- [“Exporting data” on page 25](#) describes how to export data to a text file.

Saving changes from a QueryDataSet

You can use different *Resolver* implementations to save changes back to your data source. `QueryDataSets` use a `QueryResolver` to save changes by default. The default resolver can be overridden by setting the `StorageDataSet.resolver` property. When data is provided to the data set, the `StorageDataSet` tracks the row status information (either deleted, inserted, or updated) for all rows. When data is *resolved* back to a data source like a SQL server, the row status information is used to determine which rows to add to, delete from, or modify in the SQL table. When a row has been successfully resolved, it obtains a new row status of resolved (either `RowStatus.UPDATE_RESOLVED`, `RowStatus.DELETE_RESOLVED`, or `RowStatus.INSERT_RESOLVED`). If the `StorageDataSet` is resolved again, previously resolved rows will be ignored, unless changes have been made subsequent to previous resolving.

This topic explores the basic resolver functionality provided by the DataExpress package. It extends the concepts explored in [“Querying a database” on page 46](#) to the resolving phase where you save your changes back to the data source.

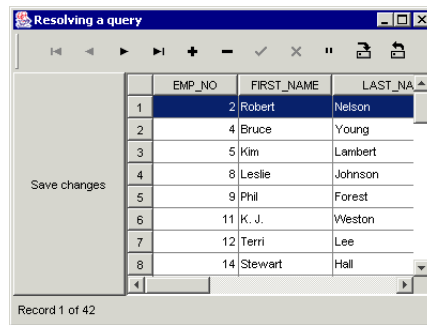
To step through this example, start with the completed sample files, located in the `/samples/DataExpress/QueryProvider` directory, or create the application by following the steps in [“Retrieving data for the examples” on page 122](#).

[“Querying a database” on page 46](#) explored the providing phase, where data is obtained from a data source. The material showed how to instantiate a `QueryDataSet` and associated UI components, and display the data retrieved from the `JDataStore` employee sample. The Save button on the `JdbNavToolBar` can be used to save data changes back to the employee file. In the next topic, we add a button that also performs basic resolving code. When either the custom button or the toolbar’s Save button is clicked, the changes made to the data in the `QueryDataSet` are saved to the employee data file using the default `QueryResolver` of the `QueryDataSet`.

Adding a button to save changes from a QueryDataSet

The source code for the completed application is located in the `/samples/DataExpress/QueryResolver` directory of your JBuilder installation. The running application looks like this:

Figure 8.1 UI for saving changes from a QueryDataSet



To create this application,

- 1 Create a simple database application, as described in [“Retrieving data for the examples” on page 122](#).

If you have already created the simple database application, just open it. If you did not complete the steps to create the application, you can access the completed project files from the `/samples/DataExpress/QueryProvider` directory of your JBuilder installation.

Note

You can save time by making backup copies of these files before modifying them since other examples in this book use the simple database application created in [“Retrieving data for the examples” on page 122](#) as a starting point.

- 2 Select the Frame file in the content pane.
- 3 Add a `JButton` component from the Swing page of the component palette. Set the button's `text` property to `Save Changes`. (See the finished application at the beginning of this example for general placement of the controls in the UI.)
- 4 Make sure the `JButton` is still selected, then click the Events tab of the Inspector. Select, then double-click the `actionPerformed()` method. This changes the focus of the IDE from the UI designer to the Source pane and displays the stub for the `actionPerformed()` method.

Add the following code to the `actionPerformed()` method:

```
try {
    database1.saveChanges(queryDataSet1);
    System.out.println("Save changes succeeded");
}
catch (Exception ex) {
    // displays the exception on the JdbStatusLabel if the
    // application includes one,
    // or displays an error dialog if there isn't
    DBExceptionHandler.handleException(ex); }
```

If you've used different names for the instances of the objects, for example, `database1`, replace them accordingly.

- 5 Run the application by selecting Run!Run Project. The application compiles and displays in a separate window. Data is displayed in a table, with a Save Changes button, the toolbar, and a status label that reports the current row position and row count.

If errors are found, an error pane appears that indicates the line(s) where errors are found. The code of the custom button is the most likely source of errors, so check that the code above is correctly entered. Make corrections to this and other areas as necessary to run the application.

When you run the application, notice the following behavior:

- Use the keyboard, mouse, or toolbar to scroll through the data displayed in the table. The status label updates as you navigate.
- You can resize the window to display more fields, or scroll using the horizontal scroll bar.

Make changes to the data displayed in the table by inserting, deleting, and updating data. You can save the changes back to the server by clicking the Save Changes button you created, or the Save button of the `JdbNavToolBar`.



Note

Because of data constraints on the employee table, the save operation may not succeed depending on the data you change. Since other edits may return errors, make changes only to the `FIRST_NAME` and `LAST_NAME` values in existing rows until you become more familiar with the constraints on this table.

Saving changes back to your data source with a stored procedure

You can use different *Resolver* implementations to save changes back to your data source. `QueryDataSets` use a `QueryResolver` to save changes by default. The default resolver can be overridden by setting the `StorageDataSet.resolver` property.

This topic explores the basic resolver functionality provided by the `DataExpress` package for `ProcedureDataSet` components. It extends the concepts explored in [Chapter 6, “Using stored procedures”](#) by exploring the different methods for saving data changes back to a data source.

In this section, the retrieving tutorial is expanded by adding basic resolving capability. With a `ProcedureDataSet` component, this can be accomplished in two ways. The following sections discuss each option in more detail.

- A button that activates basic resolving code or a `JdbNavToolBar` whose Save button also performs a basic query resolve function. See [“Saving changes using a QueryResolver” on page 80](#).
- A `ProcedureResolver` that requires special coding of the stored procedure on the database on which the data should be resolved. An example of this is available in [“Saving changes with a ProcedureResolver” on page 81](#).

Saving changes using a QueryResolver

If the `resolver` property of a `ProcedureDataSet` is not set, the default resolver is a `QueryResolver` that will generate `INSERT`, `UPDATE`, and `DELETE` queries to save the changes. The `QueryResolver` requires `tableName` and `rowID` properties to be set. This method of saving changes is demonstrated in the sample applications available as a finished project in the `<jbuilder>/samples/DataExpress/ServerSpecificProcedures/` directory.

Coding stored procedures to handle data resolution

To use a `ProcedureResolver`, you need to implement three stored procedures on the database, and specify them as properties of the `ProcedureResolver`. The three procedures are:

- `insertProcedure` is invoked for every row to be inserted in the `DataSet`. The available parameters for an invocation of an `insertProcedure` are:
 - the inserted row as it appears in the `DataSet`.
 - the optional `ParameterRow` specified in the `ProcedureDescriptor`.

The stored procedure should be designed to insert a record in the appropriate table(s) given the data of that row. The `ParameterRow` may be used for output summaries or for optional input parameters.

- `updateProcedure` is invoked for every row changed in the `DataSet`. The available parameters for an invocation of an `updateProcedure` are:
 - the modified row as it appears in the `DataSet`.
 - the original row as it was when data was provided to the `DataSet`.
 - the optional `ParameterRow` specified in the `ProcedureDescriptor`.

The stored procedure should be designed to update a record in the appropriate table(s) given the original data and the modified data. Since the original row and the modified row have the same column names, the named parameter syntax has been expanded with a way to indicate the designated data row. The named parameter “:ORIGINAL.CUST_ID” thus indicates the CUST_ID of the original data row, where “:CURRENT.CUST_ID” indicates the CUST_ID of the modified data row. Similarly, a “:parameter.CUST_ID” parameter would indicate the CUST_ID field in a `ParameterRow`.

- `deleteProcedure` is invoked for every row deleted from the `DataSet`. The available parameters for an invocation of a `deleteProcedure` are:
 - the original row as it was when data was provided into the `DataSet`.
 - the optional `ParameterRow` specified in the `ProcedureDescriptor`.

The stored procedure should be designed to delete a record in the appropriate table(s) given the original data of that row.

An example of code that uses this method of resolving data to a database follows in [“Saving changes with a ProcedureResolver” on page 81](#). In the case of InterBase, also see [“Example: Using InterBase stored procedures with return parameters” on page 83](#).

Saving changes with a ProcedureResolver

The following example shows how to save changes to your database using JBuilder’s UI designer, a `ProcedureDataSet` component, and a `ProcedureResolver`. Some sample applications referencing stored procedures on a variety of servers are available in the `/samples/DataExpress/ServerSpecificProcedures` directory.

The current project contains a `JdbNavToolBar` component. In addition to enabling you to move around the table, a toolbar provides a Save Changes button. At this point, this button will use a `QueryResolver`. Once we provide a custom resolver via a `ProcedureResolver`, the Save Changes button will call the insert, update, and delete procedures specified there instead.

At this point in the application, you can run the application and have the ability to view and navigate data. In order to successfully insert, delete, or update records, however,

you need to provide the following information on how to handle these processes. With the project open,

- 1 Select the Frame file in the content pane, then select the Design tab to activate the UI designer.
- 2 Select a `ProcedureResolver` component from the DataExpress page of the component palette on the content pane. Click in the content pane to add the component to the application.
- 3 Set the `database` property of the `ProcedureResolver` to the instantiated database, `database1` in the Inspector.
- 4 Set the `deleteProcedure` property to `DELETE_COUNTRY` as follows:
 - a Select `procedureResolver1` in the component tree and click its `deleteProcedure` property in the Inspector.
 - b Double-click in the `deleteProcedure` property value field to bring up the `DeleteProcedure` dialog.
 - c Set the `Database` property to `database1`.
 - d Click `Browse Procedures`, then double-click the procedure named `DELETE_COUNTRY`.

The following statement is written in the Stored Procedure Escape or SQL Statement field:

```
execute procedure DELETE_COUNTRY :OLD_COUNTRY
```

- e Edit this statement to be:

```
execute procedure DELETE_COUNTRY :COUNTRY
```

See the text of the procedure by using the Database Pilot (Tools|Database Pilot).

Note

Don't click `Test Procedure` because this procedure does not return a result.

- 5 Set the `insertProcedure` property to `INSERT_COUNTRY` as follows:
 - a Select, then double-click the `insertProcedure` property of the `ProcedureResolver` to open the `insertProcedure` dialog.
 - b Set the `Database` field to `database1`.
 - c Click `Browse Procedures`, then double-click the procedure named `INSERT_COUNTRY`.
 - d Edit the generated code to read:

```
execute procedure INSERT_COUNTRY :COUNTRY, :CURRENCY
```

Note

Don't click `Test Procedure` because this procedure does not return a result.

- 6 Set the `updateProcedure` property to `UPDATE_COUNTRY` as follows:
 - a Select, then double-click the `updateProcedure` property of the `ProcedureResolver` to open the `updateProcedure` dialog.
 - b Set the `Database` property to `database1`.
 - c Click `Browse Procedures`, then double-click the procedure named `UPDATE_COUNTRY`.
 - d Edit the generated code to read:

```
execute procedure UPDATE_COUNTRY :ORIGINAL.COUNTRY, :CURRENT.COUNTRY,  
:CURRENT.CURRENCY
```

Note

Don't click `Test Procedure` because this procedure does not return a result.

- 7 Select `procedureDataSet1` in the project pane. Set the `resolver` property to `procedureResolver1`.

- 8 Select `procedureDataSet1`. Set its `metaDataUpdate` property to `None`.
- 9 Choose `RunIRun Project` to run the application.

When you run the application, you can browse, edit, insert, and delete data in the table. Save any change you make with the `Save Changes` button on the toolbar. Note that in this particular example, you cannot delete an existing value in the `COUNTRY` column because referential integrity has been established. To test the `DELETE` procedure, add a new value to the `COUNTRY` column and then delete it.

Example: Using InterBase stored procedures with return parameters

An InterBase stored procedure that returns values is called differently by different drivers. The list below shows the syntax for different drivers for the following function:

```
CREATE PROCEDURE fct (x SMALLINT)
RETURNS (y SMALLINT)
AS
BEGIN
    y=2*x;
END
```

Calling `fct` procedure from different drivers:

- Visigenic and InterClient version 1.3 and earlier

```
execute procedure fct ?
```

If the procedure is called through a straight JDBC driver, the output is captured in a result set with one row. JBuilder allows the following syntax to handle output values:

```
execute procedure fct ? returning_values ?
```

JBuilder will then capture the result set and set the value into the parameter supplied for the second parameter marker.

- InterClient version 1.4 and later:

```
{call fct(?,?)}
```

where the parameter markers should be placed at the end of the input parameters.

Resolving data from multiple tables

You can specify a query on multiple tables in a `QueryDataSet` and JBuilder can resolve changes to such a `DataSet`. `SQLResolver` is able to resolve SQL queries that have more than one table reference. The metadata discovery will detect which table each column belongs to, and suggest a resolution order between the tables. The properties set by the metadata discovery are:

- `Column` - `columnName`
- `Column` - `schemaName`
- `Column` - `serverColumnName`
- `StorageDataSet` - `tableName`
- `StorageDataSet` - `resolveOrder`

The `tableName` property of the `StorageDataSet` is not set. The `tableName` is identified on a per column basis.

The property `resolveOrder` is a `String` array that specifies the resolution order for multi-table resolution. `INSERT` and `UPDATE` queries use the order of this array, `DELETE` queries use the reverse order. If a table is removed from the list, the columns from that table will not be resolved.

Considerations for the type of linkage between tables in the query

A multi-table SQL query usually defines a link between tables in the WHERE clause of the query. Depending on the nature of the link and the structure of the tables, this link may be of four distinct types (given the primary table T1 and a linked table T2):

- **1:1**

There is exactly one record in T2 that corresponds to a record in T1 and vice versa. A relational database may have this layout for certain tables for either clarity or a limitation of the number of columns per table.

- **1:M**

There can be several records in T2 that correspond to a record in T1, but only one record in T1 corresponds to a record in T2. Example: each customer can have several orders.

- **M:1**

There is exactly one record in T2 that correspond to a record in T1, but several records in T1 may correspond to a record in T2. Example: each order may have a product id, which is associated with a product name in the products table. This is an example of a lookup expressed directly in SQL.

- **M:M**

The most general case.

JBuilder takes a simplified approach to resolving multiple, linked tables: JBuilder only resolves linkages of type 1:1. However, because it is difficult to detect which type of linkage a given SQL query describes, JBuilder assumes that any multi-table query is of type 1:1. If the multiple, linked tables are not of type 1:1, you handle resolving of other types as follows:

- **1:M**

It is generally uninteresting to replicate the master fields for each detail record in the query. Instead, create a separate detail dataset, which allows correct resolution of the changes.

- **M:1**

These should generally be handled using the lookup mechanism. However if the lookup is for display only (no editing of these fields), it could be handled as a multi-table query. For at least one column, mark the `rowId` property from the table with the lookup as not resolvable.

- **M:M**

This table relationship arises very infrequently, and often it appears as a result of a specification error.

Table and column references (aliases) in a query string

A query string may include table references and column references or aliases.

- Table aliases are usually not used in single table queries, but are often used in multiple table queries to simplify the query string or to differentiate tables with the same name, owned by different users.

```
SELECT A.a1, A.a2, B.a3 FROM Table_Called_A AS A, Table_Called_B AS B
```

- Column references are usually used to give a calculated column a name, but may also be used to differentiate columns with the same name originating from different tables.

```
SELECT T1.NO AS NUMBER, T2.NO AS NR FROM T1, T2
```

- If a column alias is present in the query string, it becomes the `columnName` of the `Column` in `JBuilder`. The physical name inside the original table is assigned to the `serverColumnName` property. The `QueryResolver` uses `serverColumnName` when generating resolution queries.
- If a table alias is present in the query string, it is used to identify the `tableName` of a `Column`. The alias itself is not exposed through the `JBuilder` API.

Controlling the setting of the column properties

The `tableName`, `schemaName`, and `serverColumnName` properties are set by the `QueryProvider` for a `QueryDataSet` unless the `metaDataUpdate` property does not include `metaDataUpdate.TABLENAME`.

What if a table is not updatable?

If there is no `rowId` in a certain table of a query, all the updates to this table are not saved with the `saveChanges()` call.

Note The ability to update depends on other things, which are described in more detail in [“Querying a database” on page 46](#).

How can the user specify that a table should never be updated?

For a multi-table query, one of the tables can be updatable when the other is not. The `StorageDataSet` property `resolveOrder` is a `String` array that specifies the resolution order for multi-table resolution. `INSERT` and `UPDATE` queries use the order of this array, `DELETE` queries use the reverse order. If a table is removed from the list, the columns from that table will not be resolved.

For a single table, set the `metaDataUpdate` property to `NONE`, and do not set any of the resolving properties (`rowID`, `tableName`, etc.).

Using DataSets with RMI (streamable data sets)

Streamable data sets enable you to create a Java object (`DataSetData`) that contains all the data of a `DataSet`. Similarly, the `DataSetData` object can be used to provide a `DataSet` with column information and data.

The `DataSetData` object implements the `java.io.Serializable` interface and may subsequently be serialized using `writeObject` in `java.io.ObjectOutputStream` and read using `readObject` in `java.io.ObjectInputStream`. This method turns the data into a byte array and passes it through sockets or some other transport medium. Alternatively, the object can be passed via Java RMI, which will do the serialization directly.

In addition to saving a complete set of data in the `DataSet`, you may save just the changes to the data set. This functionality can implement a middle-tier server that communicates with a DBMS and a thin client which is capable of editing a `DataSet`.

Example: Using streamable data sets

One example of when you would use a streamable `DataSet` is in a 3-tier system with a Java server application that responds to client requests for data from certain data sources. The server may use `JBuilder QueryDataSet` components or `ProcedureDataSet` components to provide the data to the server machine. The data can be extracted using `DataSetData.extractDataSet` and sent over a wire to the client. On the client side, the data can be loaded into a `TableDataSet` and edited with `JBuilder DataSet` controls or

with calls to the `DataSet` Java API. The server application may remove all the data in its `DataSet` such that it will be ready to serve other client applications.

When the user on the client application wants to save the changes, the data may be extracted with `DataSetData.extractDataSetChanges` and sent to the server. Before the server loads these changes, it should get the physical column types from the DBMS using the metadata of the `DataSet`. Next, the `DataSet` is loaded with the changes and the usual resolvers in `JBuilder` are applied to resolve the data back to the DBMS.

If resolution errors occur, they might not be detected by UI actions when the resolution is happening on a remote server machine. The resolver could handle the errors by creating an errors `DataSet`. Each error message should be tagged with the `INTERNALROW` value of the row for which the error occurred. `DataSetData` can transport these errors to the client application. If the `DataSet` is still around, the client application can easily link the errors to the `DataSet` and display the error text for each row.

Using streamable DataSet methods

The static methods `extractDataSet` and `extractDataSetChanges` will populate the `DataSetData` with nontransient private data members, that specify

1 Metadata information consisting of

- `columnCount`
- `rowCount`
- `columnNames`
- `dataTypes`
- `rowId`, `hidden`, `internalRow` (column properties)

The properties are currently stored as the 3 high bits of each data type. Each data type is a byte. The `columnCount` is stored implicitly as the length of the `columnNames` array.

2 Status bits for each row. A `short` is stored for each row.

3 Null bits for each data element. 2 bits are stored for each data element. The possible values used are:

- 0) Normal data
- 1) Assigned Null
- 2) Unassigned Null
- 3) Unchanged Null

The last value is used only for `extractDataSetChanges`. Values that are unchanged in the UPDATED version are stored as null, saving space for large binaries, etc.

4 The data itself, organized in an array of column data. If a data column is of type `Variant.INTEGER`, an `int` array will be used for the values of that column.

5 For `extractDataSetChanges`, a special column, `INTERNALROW`, is added to the data section. This data column contains long values that designate the `internalRow` of the `DataSet` the data was extracted from. This data column should be used for error reporting in case the changes could not be applied to the target DBMS.

The `loadDataSet` method will load the data into a `DataSet`. Any columns that do not already exist in the `DataSet` will be added. Note that physical types and properties such as `sqlType`, `precision`, and `scale` are not contained in the `DataSetData` object. These properties must be found on the DBMS directly. However these properties are not necessary for editing purposes. The special column `INTERNALROW` shows up as any other column in the data set.

Customizing the default resolver logic

JBuilder makes it easy to write a custom resolver for your data when you are accessing data from a custom data source, such as EJB, application servers, SAP, BAAN, IMS, OS/390, CICS, VSAM, DB2, etc.

The retrieval and update of data from a data source, such as an Oracle or Sybase server, is isolated to two key interfaces: providers and resolvers. *Providers* retrieve data from a data source into a `StorageDataSet`. *Resolvers* save changes back to a data source. By cleanly isolating the retrieval and updating of data to two interfaces, it is easy to create new provider/resolver components for new data sources. JBuilder currently provides implementations for standard JDBC drivers that provide access to popular databases such as support for Oracle, Sybase, Informix, InterBase, DB2, MS SQL Server, Paradox, dBASE, FoxPro, Access, and other popular databases. These include:

- `OracleProcedureProvider`
- `ProcedureProvider`
- `ProcedureResolver`
- `QueryProvider`
- `QueryResolver`

An example project with a custom provider and resolver is located in the `/samples/DataExpress/CustomProviderResolver` directory of your JBuilder installation. The sample file `TestApp.java` is an application with a frame that contains a `JdbTable` and a `JdbNavToolBar`. Both visual components are connected to a `TableDataSet` component where data is provided from a custom Provider (defined in the file `ProviderBean.java`), and data is saved with a custom Resolver (defined in the file `ResolverBean.java`). This sample application reads from and saves changes to the text file `data.txt`, a simple non-delimited text file. The structure of `data.txt` is described in the interface file `DataLayout.java`.

An example describing how to write a custom `ProcedureResolver` is available in the [“Saving changes with a ProcedureResolver” on page 81](#).

Understanding default resolving

If you have not specifically instantiated a `QueryResolver` component when resolving data changes back to the data source, the built-in resolver logic creates a default `QueryResolver` component for you. This topic explores using the `QueryResolver` to customize the resolution process.

The `QueryResolver` is a `DataExpress` package component which implements the `SQLResolver` interface. It is this `SQLResolver` interface which is used by the `ResolutionManager` during the process of resolving changes back to the database. As its name implies, the `ResolutionManager` class manages the resolving phase.

Each `StorageDataSet` has a `resolver` property. If this property has not been set when you call the `Database.saveChanges()` method, it creates a default `QueryResolver` and attempts to save the changes for a particular `DataSet`.

Adding a QueryResolver component

To add a `QueryResolver` component to your application using the JBuilder visual design tools:

- 1 Open an existing project that you want to add custom resolver logic to.

The project should include a `Database` object, and a `QueryDataSet` object. See [“Querying a database” on page 46](#) for how to do this.

- 2 Double-click the `Frame` file in the content pane, and select the `Design` tab to display the UI designer.

- 3 Click the `QueryResolver` component from the DataExpress page of the component palette.
- 4 Click (anywhere) in the UI designer or the component tree to add it to your application.

The UI designer generates source code that creates a default `QueryResolver` object.

- 5 Connect the `QueryResolver` to the appropriate `DataSet`.

To do this, use the Inspector to set the `resolver` property of the `StorageDataSet`, for example `queryDataSet1`, to the appropriate `QueryResolver`, which is, by default, `queryResolver1`.

You can connect the same `QueryResolver` to more than one `DataSet` if the different `DataSet` objects share the same event handling. If each `DataSet` needs custom event handling, create a separate `QueryResolver` for each `StorageDataSet`.

Intercepting resolver events

You control the resolution process by intercepting `Resolver` events. When the `QueryResolver` object is selected in the content pane, the Events tab of the Inspector displays its events. The events that you can control (defined in the `ResolverListener` interface) can be grouped into three categories of:

- Notification of an action to be performed. Any errors will be treated as normal exceptions, not as error events.
 - `deletingRow()`
 - `insertingRow()`
 - `updatingRow()`
- Notification that an action has been performed:
 - `deletedRow()`
 - `insertedRow()`
 - `updatedRow()`
- Conditional errors that have occurred. These are internal errors, not server errors.
 - `deleteError()`
 - `insertError()`
 - `updateError()`

When the resolution manager is about to perform a delete, insert, or update action, the corresponding event notification from the first set of events (`deletingRow`, `insertingRow`, and `updatingRow`) is generated. One of the parameters passed with the notification to these events is a `ResolverResponse` object. It is the responsibility of the event handler (also referred to as the event listener) to determine whether or not the action is appropriate and to return one of the following (`ResolverResponse`) responses:

- `resolve()` instructs the resolution manager to continue resolving this row
- `skip()` instructs the resolution manager to skip this row and continue with the rest
- `abort()` instructs the resolution manager to stop resolving

If the event's response is `resolve()` (the default response), then one of the second set of events (`deletedRow`, `insertedRow` or `updatedRow`) is generated as appropriate. No response is expected from these events. They exist only to communicate to the application what action has been performed.

If the event's response is `skip()`, the current row is not resolved and the resolving process continues with the next row of data.

If the event terminates the resolution process, the `inserting` method gets called, which in turn calls `response.abort()`. No error event is generated because error events are wired to respond to internal errors. However, a generic `ResolutionException` is thrown to cancel the resolution process.

If an error occurs during the resolution processing, for example, the server did not allow a row to be deleted, then the appropriate error event (`deleteError`, `insertError`, or `updateError`) is generated. These events are passed the following:

- The original `DataSet` involved in the resolving
- A temporary `DataSet` that has been filtered to show only the affected rows
- The `Exception` which has occurred
- An `ErrorResponse` object

It is the responsibility of the error event handler to:

- examine the `Exception`
- determine how to proceed
- to communicate this decision back to the resolution manager. This decision is communicated using one of the following `ErrorResponse` responses:
 - `abort()` instructs the resolution manager to cease all resolving
 - `retry()` instructs the resolution manager to try the last operation again
 - `ignore()` instructs the resolution manager to ignore the error and to proceed

If the event handler throws a `DataSetException`, it is treated as a `ResolverResponse.abort()`. In addition, it triggers the error event described above, passing along the user's `Exception`.

Using resolver events

For an example of resolver events, see `ResolverEvents.jpx` and associated files in the `/samples/DataExpress/ResolverEvents` directory of your JBuilder installation. In the `ResolverEvents` application,

- A table is bound to the Customer table in the JDataStore sample database.
- The Save Changes button creates a custom `QueryResolver` object which takes control of the resolution process.

In the running application, you'll notice the following behavior:

- Row deletions are not allowed. Any attempt at deleting a row of data is unconditionally prevented. This demonstrates usage of the `deletingRow` event.
- Row insertions are permitted only if the customer is from the United States. If the current customer is not from the U.S., the process is aborted. This example demonstrates usage of the `insertingRow` event and a `ResolverResponse` of `abort()`.
- Row updates are done by adding the old and new values of a customer's name to a `ListControl`. This demonstrates how to access both the new information as well as the prior information during the resolution process.

Writing a custom data resolver

This topic discusses custom data resolvers, and how they can be used as resolvers for a `TableDataSet` and any `DataSet` derived from `TableDataSet`. The main method to implement is `resolveData()`. This method collects the changes to a `StorageDataSet` and resolves these changes back to the source.

In order to resolve data changes back to a source,

- 1 Make sure that the `StorageDataSet` is blocked for changes in the provider during the resolution process. This is done by calling the methods:

- `ProviderHelp.startResolution(dataSet, true);`
- `ProviderHelp.endResolution(dataSet);`

Important

Place all of the following items between these two method calls.

- 2 Locate changes in the data by creating a `DataSetView` for each of the inserted, deleted, and updated rows. That is accomplished using the following method calls:

- `StorageDataSet.getInsertedRows(DataSetView);`
- `StorageDataSet.getDeletedRows(DataSetView);`
- `StorageDataSet.getUpdatedRows(DataSetView);`

It is important to note that

- The inserted rows may contain deleted rows (which shouldn't be resolved).
- The deleted rows may contain inserted rows (which shouldn't be resolved).
- The updated rows may contain deleted and inserted rows (which shouldn't be handled as updates).

- 3 Close each of the `DataSetView` components after the data has been resolved, or if an exception occurs during resolution. If the `DataSetView` components are not closed, the `StorageDataSet` retains references to it, and such a view will never be garbage collected.

Handling resolver errors

Errors can be handled in numerous ways, however the `DataSet` must be told to change the status of the changed rows. To do this,

- 1 Change each row so that it is marked `RowStatus.PENDING_RESOLVED`:

The code to mark the current row this way is:

```
DataSet.markPendingStatus(true);
```

Call this method for each of the inserted, deleted, and updated rows that is being resolved.

- 2 Call one or more of the following methods to reset the `RowStatus.PENDING_RESOLVED` bit.

Which methods are called depends on the error handling approach:

- `markPendingStatus(false);`

The `markPendingStatus` method resets the current row.

- `resetPendingStatus(boolean resolved);`

This `resetPendingStatus` method resets all the rows in the `DataSet`.

- `resetPendingStatus(long internalRow, boolean resolved);`

This `resetPendingStatus` method resets the row with the specified `internalRow` id.

- 3 Reset the `resolved` parameter, using one of the `resetPendingStatus` methods, to **true** for rows whose changes were actually made to the data source.

When the `PENDING_RESOLVED` bit is reset, the rows retain the status of recorded changes. The rows must be reset and resolved so that,

- The `INSERTED` & `UPDATED` rows are changed to `LOADED` status.
- The `DELETED` rows are removed from the `DataSet`.

The row changes that were not made will clear the `PENDING_RESOLVED` bit, however, the changes are still recorded in the `DataSet`.

Some resolvers will choose to abandon all changes if there are any errors. In fact, that is the default behavior of `QueryDataSet`. Other resolvers may choose to commit certain changes, and retain the failed changes for error messages.

Resolving master-detail relationships

Master-detail resolution presents some issues to be considered. If the source of the data has referential integrity rules, the `DataSet` components may have to be resolved in a certain order. When using JDBC, JBuilder provides the `SQLResolutionManager` class. This class ensures the master data set resolves its inserted rows before enabling the detail data set to resolve its inserted row, and also ensures that detail data sets resolve their deleted rows before the deleted rows of the master data set are resolved. For more information on resolving master-detail relationships, see [“Saving changes in a master-detail relationship” on page 100](#).

Establishing a master-detail relationship

Databases that are efficiently designed include multiple tables. The goal of table design is to store all the information you need in an accessible, efficient manner. Therefore, you want to break down a database into tables that identify the separate entities (such as persons, places, and things) and activities (such as events, transactions, and other occurrences) important to your application. To better define your tables, you need to identify and understand how they relate to each other. Creating several small tables and linking them together reduces the amount of redundant data, which in turn reduces potential errors and makes updating information easier.

In JBuilder, you can join, or link, two or more data sets that have at least one common field using a `MasterLinkDescriptor`. A master-detail relationship is usually a one-to-many type relationship among data sets. For example, say you have a data set of customers and a data set of orders placed by these customers, where customer number is a common field in each. You can create a master-detail relationship that will enable you to navigate through the customer data set and have the detail data set display only the records for orders placed by the customer who is exposed in the current record.

You can link one master data set to several detail data sets, linking on the same field or on different fields. You can also create a master-detail relationship that cascades to a one-to-many-to-many type relationship. Many-to-one or one-to-one relationships can be handled within a master-detail context, but these kinds of relationships would be better handled through the use of lookup fields, in order to view all of the data as part of one data set. For information on saving changes to data from multiple data sets, see [“Resolving data from multiple tables” on page 83](#).

The master and detail data sets do not have to be of the same data set type. For example, you could use a `QueryDataSet` as the master data set and a `TableDataSet` as the detail data set. `QueryDataSet`, `TableDataSet`, and `DataSetView` can all be used as either master or detail data sets.

These are the topics covered:

- [“Defining a master-detail relationship” on page 94](#)
- [“Fetching details” on page 97](#)
- [“Editing data in master-detail data sets” on page 98](#)
- [“Steps to creating a master-detail relationship” on page 99](#)
- [“Saving changes in a master-detail relationship” on page 100](#)

Defining a master-detail relationship

When defining a master-detail relationship, you must link columns of the same data type. For example, if the data in the master data set is of type INT, the data in the detail data set must be of type INT as well. If the data in the detail data set were of type LONG, either no matches or incorrect matches would be found. The names of the columns may be different. You are not restricted to linking on columns that have indexes on the server.

You can sort information in the master data set with no restrictions. Linking between a master and a detail data set uses the same mechanism as maintaining sorted views, a maintained index. This means that a detail data set will always sort with the detail linking columns as the left-most sort columns. Additional sorting criteria must be compatible with the detail linking columns. To be compatible, the sort descriptor cannot include any detail linking columns or, if it does include detail linking columns, they must be specified in the same order in both the detail linking columns and the sort descriptor. If any detail linking columns are included in the sort descriptor, all of them should be specified.

You can filter the data in the master data set, the detail data set, or in both. A master-detail relationship alone is very much like a filter on the detail data set; however, a filter can be used in addition to the master-detail relationship on either data set.

Instead of using a `MasterLinkDescriptor`, you may use a SQL JOIN statement to create a master-detail relationship. A SQL JOIN is a relational operator that produces a single table from two tables, based on a comparison of particular column values (join columns) in each of the data sets. The result is a single data set containing rows formed by the concatenation of the rows in the two data sets wherever the values of the join columns compare. To update JOIN queries with JBuilder, see [“Resolving data from multiple tables” on page 83](#).

Creating an application with a master-detail relationship

This example shows how to create a master-detail relationship, using the sample files shipped with JBuilder. The basic scenario for the sample application involves constructing two queries, one that selects all of the unique countries from the COUNTRY table in the employee sample database, and one that selects all of the employees. This example is available as a finished project in the `/samples/DataExpress/MasterDetail` directory of your JBuilder installation.

The COUNTRY data set is the master data set, with the column COUNTRY being the field that we will link to EMPLOYEE, the detail data set. Both data sets are bound to `JdbTables`, and as you navigate through the COUNTRY table, the EMPLOYEE table displays all of the employees who live in the country indicated as the current record.

To create this application,

- 1 Close any open projects (File|Close).
- 2 Choose File|New, and double-click the Application icon to create a new application. Accept all defaults.
- 3 Select the Design tab in the content pane.
- 4 Select a `Database` component from the DataExpress page of the component palette, and click in the component tree or the UI designer to add the component to your application.

- 5 Open the `connection` property for the `Database` component in the Inspector, and set properties as follows, assuming your system is set up to use the `JDataStore` sample as described in “Setting up `JDataStore`” on page 31.

Property name	Value
Driver	<code>com.borland.datastore.jdbc.DataStoreDriver</code>
URL	Browse to your local copy of <code><jbuilder>/samples/JDataStore/datastores/employee.jds</code>
Username	Enter your name (the default is “SYSDBA”)
Password	Enter your password (the default is “masterkey”)

The `connection` dialog includes a Test Connection button. Click this button to check that the connection properties have been correctly set. Results of the connection attempt are displayed beside the button. When the connection is successful, click OK.

Note The `JDataStore` Server library is added to your project when you connect to a `JDataStore` database.

The code generated by the designer for this step can be viewed by selecting the Source tab and looking for the `ConnectionDescriptor` code. Click the Design tab to continue.

- 6 Select a `QueryDataSet` component from the `DataExpress` page, and click in the component tree to add the component to your application.

This component sets up the query for the master data set. Select the `query` property of the `QueryDataSet` component from the Inspector, and set as follows:

Property name	Value
Database	<code>database1</code>
SQL Statement	<code>SELECT * FROM COUNTRY</code>

- 7 Click Test Query to ensure that the query is runnable, and when the status area indicates `Success`, click OK to close the dialog box.
- 8 Add another `QueryDataSet` component to your application, select its `query` property in the Inspector, click the ellipsis (...) button to open the Query dialog box, and set the following properties:

Property name	Value
Database	<code>database1</code>
SQL Statement	<code>SELECT * FROM EMPLOYEE</code>

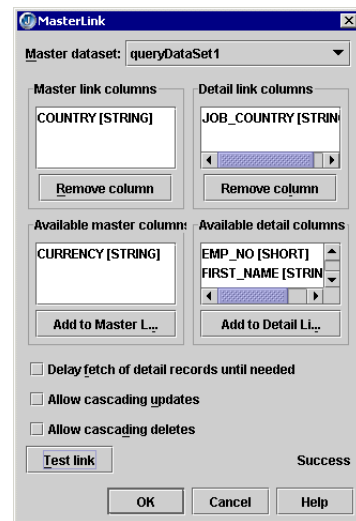
This will set up the query for the detail data set.

- 9 Click Test Query to ensure that the query is runnable, and when the status area indicates `Success`, click OK to close the dialog box.
- 10 Select the `masterLink` property for the detail data set (`queryDataSet2`) in the Inspector, click the ellipsis (...) button to open the MasterLink dialog box, and set the properties as follows:
- a The `Master DataSet` property provides a drop-down menu of available data sets. Choose the data set that contains the master records for the current detail data set, in this case select `queryDataSet1`.
 - b The link fields describe which fields to use when determining matching data between the master and detail data set components. To select a column from the master data set to link with a column in the detail data set, select the column name, in this case `COUNTRY` (a string field), from the list of Available Master

Columns then click the Add to Master Links button. This column displays in the Master Link Columns box.

- c To select the column from the detail data set to link with a column in the master data set, select the column name, in this case JOB_COUNTRY (a string field), from the list of Available Detail Columns, then click the Add to Detail Links button. This column displays in the Detail Link Columns box.
- d The Delay Fetch Of Detail Records Until Needed option determines whether the records for the detail data set can be fetched all at once or can be fetched for a particular master when needed (when the master record is visited). Uncheck this box to set `fetchAsNeeded` to `false`. For more information on fetching, see [“Fetching details” on page 97](#).
- e Click Test Link.

The dialog should look like this when the test is successful.



- f Click OK to close the MasterLink dialog box.

- 11 Add a `DBDisposeMonitor` to your application from the More dbSwing page.

The `DBDisposeMonitor` will close the `JDataStore` when the window is closed.

- 12 Set the `dataAwareComponentContainer` property of the `DBDisposeMonitor` to this.

To create a UI for this application,

- 1 Select `contentPane` (`BorderLayout`) in the component tree, and set its `layout` property to `null`.
- 2 Add a `JdbNavToolBar` component from the dbSwing page, and drop the component in the area at the top of the panel in the UI Designer.
- 3 Add a `JdbStatusLabel` and drop it in the area at the bottom of the panel in the UI designer.

`JdbNavToolBar` automatically attaches itself to whichever `DataSet` has focus.

`JdbStatusLabel` automatically attaches itself to whichever `DataSet` has focus.

- 4 Select a `TableScrollPane` component from the dbSwing page, click and drag the outline for the pane in the upper portion of the UI designer to add it to the application just under `jdbNavToolBar1`.

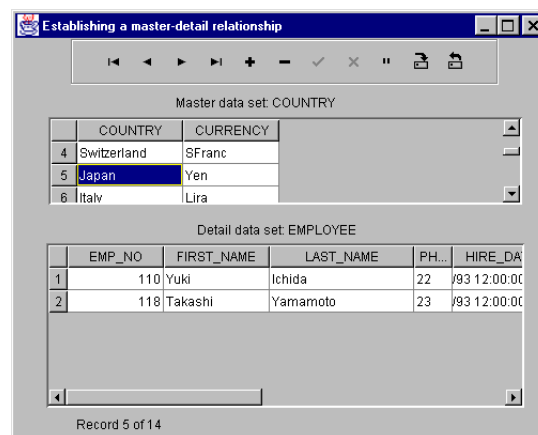
Scrolling behavior is not available by default in any Swing component or dbSwing extension, so, to get scrolling behavior, we add the scrollable Swing or dbSwing components to a `JScrollPane` or a `TableScrollPane`. `TableScrollPane` provides special

capabilities to `JdbTable` over `JScrollPane`. See the `dbSwing` documentation for more information.

- 5 Drop a `JdbTable` into the center of `tableScrollPane1` in the UI designer, and set its `dataSet` property to `queryDataSet1`.
- 6 Add another `TableScrollPane` to the lower part of the panel in the UI designer. This will become `tableScrollPane2`.
- 7 Drop a `JdbTable` into `tableScrollPane2`, and set its `dataSet` property to `queryDataSet2`.
- 8 Compile and run the application by choosing `Run!Run Project`.

Now you can move through the master (`COUNTRY`) records and watch the detail (`EMPLOYEE`) records change to reflect only those employees in the current country.

The running application looks like this:



Fetching details

In a master-detail relationship, the values in the master fields determine which detail records will display. The records for the detail data set can be fetched all at once or can be fetched for a particular master when needed (when the master record is visited).

Fetching all details at once

When the `fetchAsNeeded` parameter is `false` (or `Delay Fetch Of Detail Records Until Needed` is unchecked in the `masterLinkDescriptor` dialog box), all of the detail data is fetched at once. Use this setting when your detail data set is fairly small. You are viewing a snapshot of your data when you use this setting, which will give you the most consistent view of your data. When the `refresh()` method is called, all of the detail sets are refreshed at once.

For example, initially the data set is populated with all of the detail data set data. When the `fetchAsNeeded` option is set to `false`, you could instantiate a `DataSetView` component, view the detail data set through it, and see that all of the records for detail data set are present, but are being filtered from view based on the linking information being provided from the master data set.

Fetching selected detail records on demand

When the `fetchAsNeeded` parameter is `true` (or `Delay Fetch Of Detail Records Until Needed` is checked in the `masterLinkDescriptor` dialog box), the detail records are fetched on demand and stored in the detail data set. This type of master-detail

relationship is really a parameterized query where the values in the master fields determine which detail records will display. You are most likely to use this option if your remote database table is very large, in order to improve performance (not all of the data set will reside in memory, but it will be loaded as needed). You would also use this option if you are not interested in most of the detail data. The data that you view will be fresher and more current, but not be as consistent a snapshot of your data as when the `fetchAsNeeded` parameter is `false`. You will fetch one set of detail records at one point in time, it will be cached in memory, then you will fetch another set of detail records and it will be cached in memory. In the meantime, the first set of detail records may have changed in the remote database table, but you will not see the change until you refresh the details. When the `refresh()` method is called, only the current detail sets are refreshed.

For example, initially, the detail data set is empty. When you access a master record, for example Jones, all of the detail records for Jones are fetched. When you access another master record, say Cohen, all of the detail records for Cohen are fetched and appended to the detail data set. If you instantiate a `DataSetView` component to view the detail data set, all records for both Jones and Cohen are in the detail data set, but not any records for any other name.

When the `fetchAsNeeded` property is `true`, the query for the detail dataset must contain a WHERE clause that defines the relationship of the detail columns to a parameter that represents the value of a column in the master data set. If the parameterized query has named parameter markers, the name must match a name in the master data set. If “?” JDBC parameter markers are used, the detail link columns are bound to the parameter markers from left to right as defined in the `masterLink` property. The binding of the parameter values is implicit when the master navigates to a row for the first time. The query will be re-executed to fetch each new detail group. If there is no WHERE clause, JBuilder throws `DataSetException.NO_WHERE_CLAUSE`. When fetching is handled this way, if no explicit transactions are active, the detail groups will be fetched in separate transactions. For more information on master-detail relationships within parameterized queries, see [“Parameterized queries in master-detail relationships” on page 59](#).

When the master data set has two or more detail data sets associated with it, and the `fetchAsNeeded` property of each is `true`, the details remember what detail groups they have attempted to fetch via a query or stored procedure that is parameterized on the active master row linking columns. This memory can be cleared by calling the `StorageDataSet.empty()` method. There is no memory for `masterLink` properties that do not set `fetchAsNeeded` to `true`.

When the detail data set is a `TableDataSet`, the `fetchAsNeeded` parameter is ignored and all data is fetched at once.

Editing data in master-detail data sets

By default, you cannot delete or change a value in a master link column (a column that is linked to a detail data set) if the master record has detail records associated with it. Also by default, detail link columns will not be displayed in a `JdbTable` UI component, because these columns duplicate the values in the master link columns, which are displayed. When a new row is inserted into the detail data set, JBuilder will insert the matching values in the non-displayed fields.

Using the `cascadeUpdates` and `cascadeDeletes` properties for master-detail relationships, you can enable updates to rows in a detail `DataSet` based on updates in linking columns in the master `DataSet`, and deletions of matching rows in a detail `DataSet` when corresponding rows are deleted in the master `DataSet`. These properties can be set in the MasterLink property editor with the Allow Cascading Updates and Allow Cascading Deletes checkboxes.

Be careful when using the `cascadeUpdates` and `cascadeDeletes` options for master-detail relationships, especially in cases where you have multiple master-detail relationships

chained, such that a detail `DataSet` is a master to another detail (and so on). When using these options, one row of a detail data set may be updated or deleted, but the others may not. For example, an event handler for the `deleting()` event of the `editListener` may allow deletion of some detail rows and block deletion of others. In the case of cascaded updates, you may end up with orphan details if some rows in a detail set can be updated and others cannot. For more information on the `cascadeUpdates` and `cascadeDeletes` options, see the `MasterLinkDescriptor` topic in the *DataExpress Component Library Reference*.

Steps to creating a master-detail relationship

To create a master-detail link between two data set components, one which represents the master data set and another which represents the detail data set,

- 1 Create or open an application with at least two data set components, one of which represents the master data set and another which represents the detail data set.

You can use the Master-detail sample application in the `MasterDetail.jpx` project located in the `<jbuilder>/samples/DataExpress/MasterDetail` directory.

- 2 Select the Frame file in the content pane. Select the Design tab to activate the UI designer.
- 3 Select the detail data set in the component tree, and select its `masterLink` property from the Properties page of the Inspector. In the `masterLink` custom property editor, specify the following properties for the detail data set:
 - The `masterDataSet` property provides a choice menu of available data sets. Choose the data set that contains the master records for the current detail data set.
 - The link columns describe which columns to use when determining matching data between the master and detail data set components. To select a column from the master data set to link with a column in the detail data set, double-click the column name in the list of `Available Master Columns`. This column will now display in the `Master Link Columns` property.
 - To select the column of the detail data set to link with a column in the master data set, double-click the column name from the list of `Available Detail Columns`. The data type for each column is shown. If you select a detail column whose type does not match the corresponding master column, nothing will happen since the link columns must match by type. When properly selected, this column will display in the `Detail Link Columns` property.
 - To link the two data sets on more than one column, repeat the previous two steps until all columns are linked.
 - To delay fetching detail records until they are needed, check the `Delay Fetch Of Detail Records Until Needed` box. See [“Fetching details” on page 97](#) for more discussion on this option.
 - To verify that the data sets are properly connected, click `Test Link`. The status area will indicate `Running`, `Success`, or `Failed`.
 - To complete the specification, click `OK`.
- 4 Add visual components (such as `JdbTables`) to enable you to view and modify data. Set the `dataSet` property of one to the master data set, and set the `dataSet` property of the other to the detail data set.
- 5 Compile and run the application.

The master data set will display all records. The detail data set will display the records that match the values in the linked columns of the current row of the master data set, but (by default) will not display the linked columns.

Saving changes in a master-detail relationship

In JBuilder, data is retrieved from a server or text file into a data set. Once this data has been “provided” to the data set, you can edit and work with a local copy of the data programmatically or in data-aware components. To save the data back to the database or text file, you must “resolve” the changes back to the database or export the changes to a text file. The different options for resolving the changes back to the database are discussed in [Chapter 8, “Saving changes back to your data source”](#) and the options for exporting data to a text file are discussed in [“Exporting data” on page 25](#).

In a master-detail relationship, at least two sets of data (database tables and/or text data files in any combination) are being provided to at least two data sets. In general, there are three ways you can resolve changes in a master-detail relationship:

- Place a `JButton` in your application and write the resolver code for the button that commits the data for each data set. An example of this can be found in the topic [“Saving changes from a QueryDataSet” on page 78](#).

If both data sets are `QueryDataSets`, you can save changes in both the master and the detail tables using the `saveChanges(DataSet[])` method of the `Database` rather than the `saveChanges()` method for each data set. Using a call to the `Database.saveChanges(DataSet[])` method keeps the data sets in sync and commits all data in one transaction. Using separate calls to the `DataSet.saveChanges()` method does not keep the data sets in sync and commits the data in separate transactions. See [“Resolving master-detail data sets to a JDBC data source” on page 100](#) for more information.

- Place a `QueryResolver` in your application to customize resolution. See [“Customizing the default resolver logic” on page 87](#) for more information.
- Place a `JdbNavToolBar` in your application and use the Save button to save changes.

You can use a single `JdbNavToolBar` for both data sets. The `JdbNavToolBar` component automatically attaches itself to whichever `DataSet` has focus.

See also

- [Chapter 8, “Saving changes back to your data source”](#)

Resolving master-detail data sets to a JDBC data source

Because a master-detail relationship by definition includes at least two sets of data, the simplest way to resolve data back to the data source is to use the `saveChanges(DataSet[])` method of the `Database` component (assuming that `QueryDataSet` components are used).

Executing the `Database.saveChanges(DataSet[])` method causes all of the inserts, deletes, and updates made to the data sets to be saved to the JDBC data source in a single transaction, by default. When the `masterLink` property has been used to establish a master-detail relationship between two data sets, changes across the related data sets are saved in the following sequence:

- 1 Deletes
- 2 Updates
- 3 Inserts

For deletes and updates, the detail data set is processed first. For inserts, the master data set is processed first.

If an application is using a `JdbNavToolBar` for save and refresh functionality, the `fetchAsNeeded` property should be set to `false` to avoid losing unsaved changes. This is because when the `fetchAsNeeded` property is `true`, each detail set is fetched individually, and is also refreshed individually. Since the Save button only saves changes to the

data set that has focus, you would have to click the Save button twice, once (when the master has focus) to save changes in the master data set, and again (when the detail has focus) to save changes in the detail data set. If the `Database.saveChanges(DataSet[])` method is used instead, all edits will be posted in the right order and in the same transaction to all linked data sets.

Chapter 10

Using data modules to simplify data access

A data module is a specialized container for data access components. Data modules simplify data access development in your applications. Data modules offer you a centralized container for all your data access components. This enables you to modularize your code and separate the database access logic and business rules in your applications from the user interface logic in the application. You can also maintain control over the use of the data module by delivering only the class files to application developers.

Once you define your `DataSet` components and their corresponding `Column` components in a data module, all applications that use the module have consistent access to the data sets and columns without requiring you to recreate them in every application each time you need them. Data modules do not need to reside in the same directory or package as your project. They can be stored in a location for shared use among developers and applications.

`DataModule` is an interface which declares the basic behavior of a data module. To work with this interface programmatically, implement it in your data module class and add your data components.

When you create a data module and add any component that would automatically appear under the Data Access section of the content pane (`Database`, `DataSet`, `DataStore`), a `getter()` method is generated. This means that any of these components will be available in a choice list for the project that references the data module. This means, for example, that you can

- Add a `Database` component to a data module.
- Compile the data module.
- Add a `QueryDataSet` component to the application that contains the data module or to the data module itself.
- In the `query` property dialog, select “DataModule1.database1” (or something similar) from the Database choice box.

This chapter discusses two ways to create a data module:

- Using the JBuilder visual design tools
- Using the Data Modeler

Creating a data module using the design tools

The following sections describe how to create a data module, using the visual design tools, such as the Data Module wizard and the UI designer.

Create the data module with the wizard

To create a data module,

- 1 Create a new project.
- 2 Choose File|New and double-click the Data Module icon.
- 3 Specify the package and class name for your data module class.

JBuilder automatically fills in the Java file name and path based on your input. To create the data module using the JBuilder designer, deselect Invoke Data Modeler.

- 4 Click OK to close the dialog box.

The data module class is created and added to the project.

- 5 Double-click the data module file in the project pane to open it in the content pane.
- 6 View the source code.

You'll notice that the code generated by the wizard for the data module class is slightly different than the code generated by other wizards. The `getDataModule()` method is defined as **public static**. The purpose of this method is to allow a single instance of this data module to be shared by multiple frames. The code generated for this method is:

```
public static DataModule1 getDataModule() {
    if (myDM == null){
        myDM = new DataModule1();
    }
    return myDM;
}
```

The code for this method,

- Declares this method as **static**. This means that you are able to call this method without a current instantiation of a `DataModule` class object.
- Returns an instance of the `DataModule` class.
- Checks to see if there is a current instantiation of a `DataModule`.
- Creates and returns a new `DataModule` if one doesn't already exist.
- Returns a `DataModule` object if one has been instantiated.

The data module class now contains all the necessary methods for your custom data module class, and a method stub for the `jbInit()` to which you add your data components and custom business logic.

Add data components to the data module

To customize your data module using the UI designer,

Note Although data modules do not show up as visible components in the designer like `dbSwing` components, it is useful to work with data modules in the designer. Using the designer, you can quickly add and modify related data access components using the component palette, structure pane, and Inspector.

- 1 Double-click the data module file in the project pane to open it in the content pane.
- 2 Select the Design tab of the content pane to activate the UI designer.

3 Add your data components to your data module class.

For example,

- a Select a `Database` component from the DataExpress page of the component palette.
- b Click in the component tree or the UI designer to add the `Database` component to the `DataModule`.
- c Set the `connection` property using the database `connectionDescriptor`. Setting the `connection` property in the Inspector is discussed in [Chapter 4, "Connecting to a database."](#)

The data components are added to a data module just as they are added to a Frame file. For more information on adding data components, see [Chapter 5, "Retrieving data from a data source."](#)

Note JBuilder automatically creates the code for a public method that "gets" each `DataSet` component you place in the data module. This allows the `DataSet` components to appear as (read-only) properties of the `DataModule`. This also allows `DataSet` components to be visible to the `dataSet` property of data-aware components in the Inspector when data-aware component and data modules are used in the same container.

After you have completed this section, your data module file will look similar to this:

```
package datamoduleexample;

import com.borland.dx.dataset.*;
import com.borland.dx.sql.dataset.*;

public class DataModule1 implements DataModule{
    private static DataModule1 myDM;
    Database databasel = new Database();
    public DataModule1() {
        try {
            jbInit();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
    private void jbInit() throws Exception{
        databasel.setConnection(new
            com.borland.dx.sql.dataset.ConnectionDescriptor("
                jdbc:borland:dslocal:/usr/local/<jbuilder>/samples/JDataStore/
                datastores/employee.jds", "your name", "", false,
                "com.borland.datastore.jdbc.DataStoreDriver"));
    }
    public static DataModule1 getDataModule() {
        if (myDM == null)
            myDM = new DataModule1();
        return myDM;
    }
    public com.borland.dx.sql.dataset.Database getDatabasel() {
        return databasel;
    }
}
```

Adding business logic to the data module

Once the data components are added to the data module and corresponding properties set, you can add your custom business logic to the data model. For example, you may want to give some users the rights to delete records and not give these rights to others. To enforce this logic, you add code to various events of the `DataSet` components in the data module.

Note The property settings and business logic code you add to the components in the data model cannot be overridden in the application that uses the data model. If you have behavior that you do not want to enforce across all applications that use this data model, consider creating multiple data models that are appropriate for groups of applications or users.

To add code to the events of a component,

- 1 Double-click the data module file in the project pane to open it in the content pane.
- 2 Select the Design tab of the content pane to activate the UI designer.
- 3 Select the component to which you want to add business logic, then click the Events tab in the Inspector.
- 4 Double-click the event where you want the business logic to reside.

JBuilder creates a stub in the Java source file for you to add your custom business logic code.

Using a data module

To use a data module in your application, it must first be saved and compiled. In your data module,

- 1 Choose File|Save All.

Note the name of the project, the package, and the data module.

- 2 Compile the data module class by choosing Run|Make Project.

This creates the data module class files in the directory specified in Project|Project Properties, Output Path.

- 3 Choose File|Close.

To reference the data module in your application, you must first add it to your project as a required library.

Adding a required library to a project

These general instructions for adding a required library use a data module as a specific example, but the same steps can be used to add any required library. A library could be a class file, such as a data module, or an archive, such as a JAR (`.jar`) file.

To add a data module as a required library,

- 1 Open the Project Properties dialog box (Project|Project Properties).
- 2 Select the Required Libraries tab on the Paths page, and add the class or archive file for the new library.

In the specific case of adding a data module, this will be the data module class file you just compiled.

- 3 Click Add to open the Select One Or More Libraries dialog box.
- 4 Click New.

The New Library wizard opens.

- 5 Enter the name for the library (like Employee Data Module).
- 6 Select the location where you want your `<library name>.library` file to go.
 You have a choice between Project, User Home, and JBuilder. If you are running JBuilder from a network, and you want your library to be accessible to everyone, you should select JBuilder. This will put your `<library name>.library` file in your `/lib` directory within your JBuilder installation. If you are the only developer that needs access to your library, you may want to choose one of the other options, so the `.library` file will be stored locally.
- 7 Click Add, browse to the folder containing which contains the path to the class file or archive you wish to add, and click OK.
 JBuilder automatically determines the paths to class files, source files, and documentation within this folder. The Library Paths field should list the path to the new library.
- 8 Click OK to close the wizard.
- 9 Click OK to close the Select One Or More Libraries dialog box.
 At this point you should see your new library added to the list of required libraries.
- 10 Click OK to close the Project Properties dialog box.

Referencing a data module in your application

Now that you have added the data module as a required library, here are the remaining steps for referencing a data module in your application:

- 1 Close any open projects (File|Close Projects).
- 2 Choose File|New and double-click Application on the General page of the object gallery.
 This will create both an application and a project.
- 3 Enter the appropriate package and class information.
- 4 Select the application's Frame file in the content pane.
- 5 Make sure DataExpress is specified as one of the required libraries.
 If DataExpress is not listed under Required Libraries in the Project Properties dialog box (Project|Project Properties),
 - a Click Add.
 - b Select DataExpress.
 - c Click OK until the Project Properties dialog is closed.
- 6 Make sure the data module is added as a required library.
- 7 Import the package that the data module class belongs to (if it is outside your package) by choosing Wizards|Edit|Use DataModule.
- 8 Click the ellipsis (...) button to open the Select DataModule dialog box.
 A tree of all known packages and classes is displayed. Browse to the location of the class files generated when the data module was saved and compiled (this should be under a node of the tree with the same name as your package, if the data module is part of a package). Select the data module class. If you do not see the data module class here, check to make sure the project compiled without errors and that it was properly added to the required libraries for the project.
- 9 Click OK.
 If you get an error message at this point, double check the required libraries in the project properties, and the location of the class file for your data module.

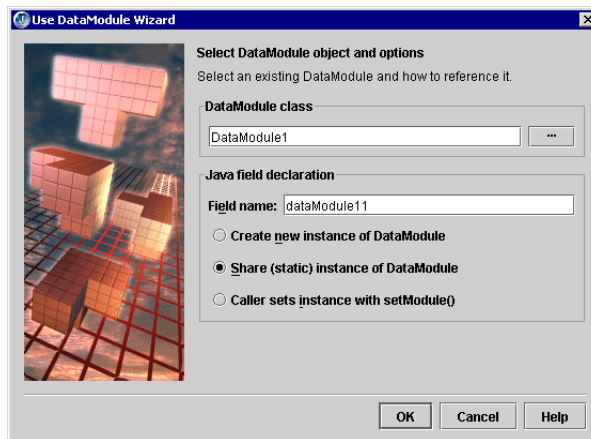
Click the Design tab to open the UI designer; the instance of the data module appears in the component tree. Clicking the entry for the data module does not display its `DataSet` components nor its `Column` components. This is intentional to avoid modification of the business logic contained in the data module from outside.

When designing your application, you'll notice that the `dataSet` property of a UI component includes all the `DataSetView` and `StorageDataSet` components that are included in your data module. You have access to them even though they are not listed separately in the content pane.

If you have a complex data model and/or business logic that you don't want another developer or user to manipulate, encapsulating it in a reusable component is an ideal way to provide access to the data but still enforce and control the business logic.

Understanding the Use DataModule wizard

When you choose `Wizards|Edit|Use DataModule`, you will see the following wizard:



Select a data module by clicking the ellipsis (...) button for the DataModule Class field. A tree of all known packages and classes is displayed. If you do not see your `DataModule` class in this list, use `Project|Project Properties` to add the data module package or archive to your required libraries for the project. Browse to the location of the class files generated when the data module was saved and compiled. Select the data module class.

In the Java Field Declaration box, the default field name is the name of the data module, followed by a "1". It is the name which will be used for the member variable to generate in code. The data module will be referred to by the name given in the component tree. Select a name that describes the data in the data module, such as `EmployeeDataModule`.

In the Java Field Declaration section, you can choose from the following ways of using the `DataModule` in your application:

- **Create New Instance Of DataModule**
If you only have a single `Frame` subclass in your application, select this option.
- **Share (Static) Instance Of DataModule**
If you plan to reference the data module in multiple frames of your application, and want to share a single instance of the custom `DataModule` class, select this option.
- **Caller Sets Instance With setModule()**
Select this option when you have several different data modules, for instance, a data module that gets the data locally and one that gets the data remotely.

Click OK to add the data module to the package and inject the appropriate code into the current source file to create an instance of the data module.

Based on the choices shown in the dialog above, the following code will be added to the `jbInit()` method of the Frame file. Note that Share (Static) Instance of Data Module is selected:

```
dataModule12 = com.borland.samples.dx.datamodule.DataModule1.getDataModule();
```

If Create New Instance Of DataModule is selected, the following code will be added to the `jbInit()` method of the Frame file:

```
dataModule12 = new com.borland.samples.dx.datamodule.DataModule1();
```

If Caller Sets Instance With SetModule() is selected, a `setModule()` method is added to the class being edited.

Creating data modules using the Data Modeler

The JBuilder IDE provides tools that can help you quickly create applications that query a database. The Data Modeler can build data modules that encapsulate a connection to a database and the queries to be run against the database. The Data Module Application wizard can then use that data module to create a client-server database application.

Creating queries with the Data Modeler

JBuilder can greatly simplify the task of viewing and updating your data in a database. The JBuilder Data Modeler lets you visually create SQL queries and save them in JBuilder Java data modules.

To begin a new project,

- 1 Choose File/New Project to start the Project wizard.
- 2 Choose a location and name for the project.
- 3 Click the Finish button.

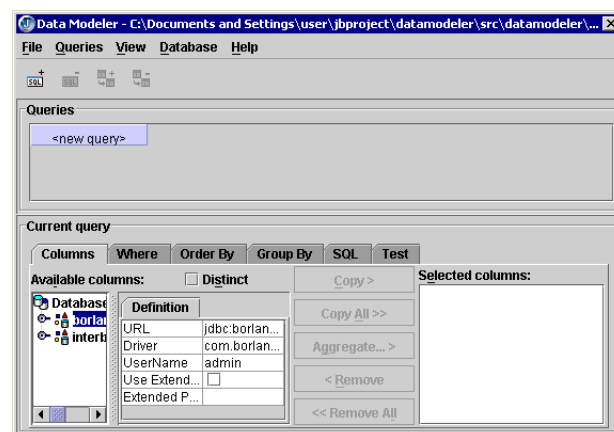
For more specific information about creating projects, see “Creating and managing projects” in *Building Applications with JBuilder*.

To display the Data Modeler,

- 1 Choose File/New.
- 2 Double-click the Data Module icon on the General page of the object gallery.
- 3 Enter the package and class name for the data module you are creating, and check the Invoke Data Modeler option.
- 4 Click OK.

The Data Modeler displays.

Figure 10.1 Data Modeler



To open an existing Java data module in the Data Modeler,

- 1 Right-click the module in the project pane.
- 2 Choose Open With Data Modeler.

Opening a URL

To begin building a SQL query, you must first open a connection URL. There are several ways you can do this:

- Double-click the URL that accesses your data.
- Choose the expand icon.
- Select the URL and choose Database|Open Connection URL.

If the database you want to access is not listed under Database URLs in the Data Modeler, you can add it.

- 1 Choose Database|Add Connection URL to display the New URL dialog box.
- 2 Select an installed driver in the drop-down Driver list or type in the driver you want.

For the samples, you can select `com.borland.datastore.jdbc.DataStoreDriver`.

- 3 Type in the URL or use the Browse button to select the URL of the data you want to access.

For the samples, you can select the `employee.jds` database located under the samples directory of your JBuilder installation, `/samples/JDataStore/datastores/employee.jds`. You can use the Browse button to browse to this file to reduce the chance of making a typing error.

Beginning a query

Begin building a query by doing the following:

- 1 Select the columns you want to add to the query from a table, or select an aggregate function that operates on a specific column.
 - a Double-click the Tables node or choose the Tables expand icon to view the tables.
 - b From the list of tables, select the table you want to query and double-click it. Double-click the Columns node to view all the columns in the selected table.
- 2 Add one or more columns to a query's SELECT statement.

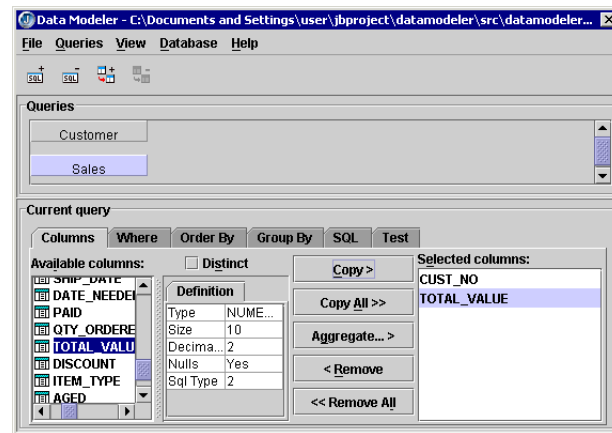
The SELECT statement is the data retrieval statement that returns a variable number of rows of a fixed number of columns. The Data Modeler helps you build the SELECT statement. The SELECT clause specifies the list of columns to be retrieved.

- a Select a column you want to add from the table you want to access.
- b Click the Copy button.

The name of the selected column appears in the Selected Columns box and the table name appears in the Queries panel at the top. Continue selecting columns

until you have all you want from that table. If you want to select all columns, click the Copy All button.

Figure 10.2 Selecting columns

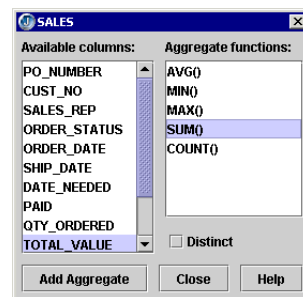


3 Add an aggregate function to the query.

Aggregate functions provide a summary value based on a set of values. Aggregate functions include SUM, AVG, MIN, MAX, and COUNT.

- a Click the Aggregate button to display a dialog box.

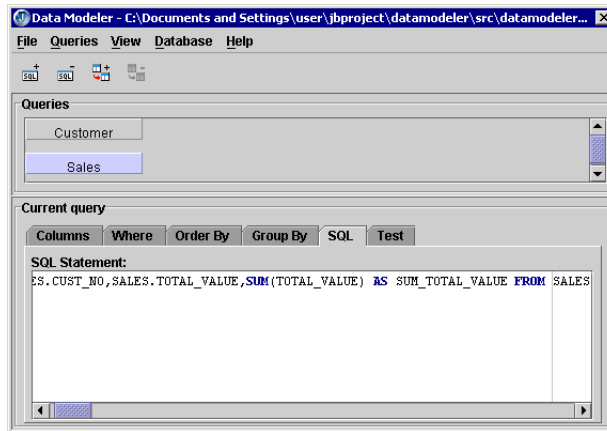
Figure 10.3 Aggregate dialog box



- b Click the column whose data values you want aggregated in the Available Columns list.
- c Click the function you want to use on that column from the Aggregate Functions column.
- d Check the Distinct check box if you want the function to operate on only unique values of the selected column.
- e Choose Add Aggregate to add the function to your query.

As you select columns and add functions, your SQL SELECT statement is being built. When you aggregate data, you must include a GROUP BY clause. For information on GROUP BY clauses, see [“Adding a Group By clause” on page 112](#).

To view your SQL statement, click the SQL tab.



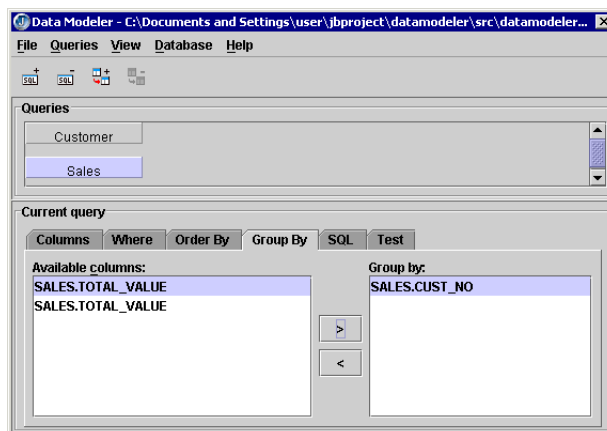
Adding a Group By clause

The GROUP BY clause is used to group data returned by a select statement and is often used in conjunction with aggregate functions. When used with aggregate functions, the following process is followed:

- First, the data is restricted by a WHERE clause, if one exists.
- Data is grouped by the field indicated in the GROUP BY clause.
- Aggregate functions are applied to the groups and a summary row is produced (one for each group).

To add a Group By clause to your query, click the Group By tab to display the Group By page.

Figure 10.4 Group By page



The Available Columns box lists the columns of the currently selected query in the Queries panel of the Data Modeler. The Group By box contains the column names the query will be grouped by. By default, the query is not grouped by any column until you specify one.

To add a Group By clause to your query,

- 1 Select the column you want the query grouped by.
- 2 Click the Add (>) button to transfer the column name to the Group By box.

A Group By clause is then added to your SQL SELECT statement. To view it, click the SQL tab.

Selecting rows with unique column values

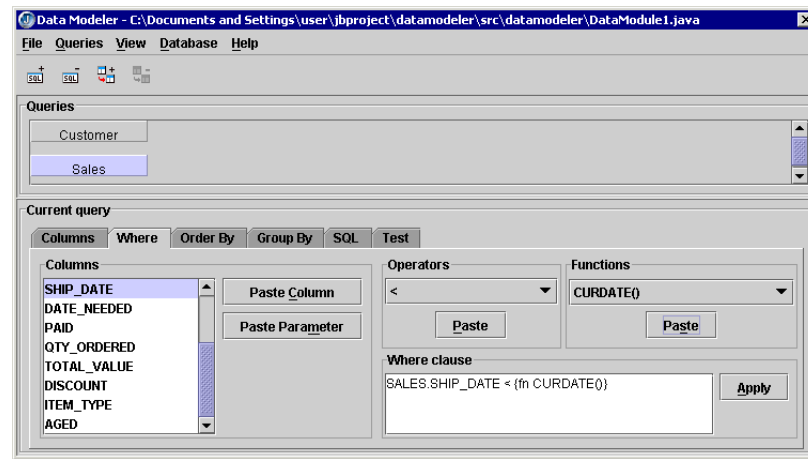
You might want to see only those rows that contain unique column values. If you add the **DISTINCT** keyword to the **SELECT** statement, the query's result set will never contain two rows with the same values in all fields. **DISTINCT** affects all columns in the **SELECT** statement.

To add the **DISTINCT** keyword, check the **Distinct** option on the **Columns** page.

Adding a Where clause

Adding a **WHERE** clause to a select statement specifies the search condition that has to be satisfied for rows to be included in the result table. To add a **Where** clause to your SQL query, click the **Where** tab.

Figure 10.5 Where page



The **Columns** list on the left contains the columns of tables in the currently selected query in the **Queries** panel of the Data Modeler.

Use the **Columns**, **Operators**, and **Functions** lists to build the clause of the query in the **Where Clause** box as follows:

- 1 Select a column in the **Columns** list and click the **Paste Column** button to transfer a column as a column name to the **Where Clause** box.
- 2 Select a column in the **Columns** list and click the **Paste Parameter** button to transfer a column as a parameter as in a parameterized query.
- 3 Select the operator you need in the **Operators** drop-down list and click the **Paste** button.

Every **Where** clause requires at least one operator.

- 4 Select the function you need in the **Functions** drop-down list if your query requires a function and click the **Paste** button.

By pasting selections, you are building a **Where** clause. You can also directly edit the text in the **Where Clause** box to complete your query. For example, suppose you are building a **Where** clause like this:

```
WHERE COUNTRY='USA'
```

You would select and paste the **COUNTRY** column and the **=** operator. To complete the query, you would type in the data value directly, which in this case is **'USA'**.

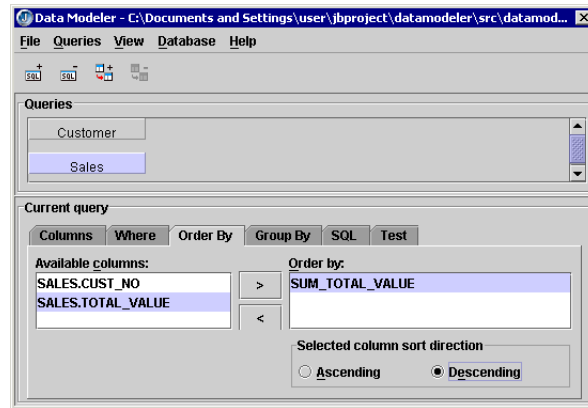
When you are satisfied with your **Where** clause, click the **Apply** button. The **Where** clause is added to the entire **SQL SELECT** statement. To view it, click the **SQL** tab.

Adding an Order By clause

An ORDER BY clause is used to sort or rearrange the order of the data in the result table. To specify how rows of a table are sorted,

- 1 Select the query you want sorted in the Queries panel.
- 2 Click the Order By tab in the Current Query panel.

Figure 10.6 Order By page



- 3 Select the column you want the query sorted by in the Available Columns box and click the Add (>) button to transfer that column to the Order By box.
- 4 Select the sort order direction from the Selected Sort Order Direction options.

The Ascending option sorts the specified column from the smallest value to the greatest, while the Descending option sorts the specified column from the greatest value to the smallest. For example, if the sort column is alphabetical, Ascending sorts the column in alphabetical order and Descending sorts it in reverse alphabetical order.

You can sort the query by multiple columns by transferring more than one column to the Order By box. Select the primary sort column first, then select the second, and so on. For example, if your query includes a Country column and a Customer column and you want to see all the customers from one country together in your query, you would first transfer the Country column to the Order By box, then transfer the Customer column. The sort direction, Ascending or Descending, can be specified for each column included in the Order By box.

Editing the query directly

At any time while you are using the Data Modeler to create your query, you can view the SQL SELECT statement and edit it directly.

To view the SELECT statement, click the SQL tab. To edit it, make your changes directly in the SELECT statement.

Testing your query

You can view the results of your query in the Data Modeler. The query created in this topic will not execute, the topics were presented in a way that made them most understandable, but not in a way that enabled the query to run properly.

To see the results of the query you are building,

- 1 Click the Test tab.
- 2 Click the Execute Query button.

If your query is a parameterized query, a Specify Parameters dialog box appears so you may enter the values for each parameter. When you choose OK, the query executes and you can see the results. The values you entered are not saved in the data module.

Building multiple queries

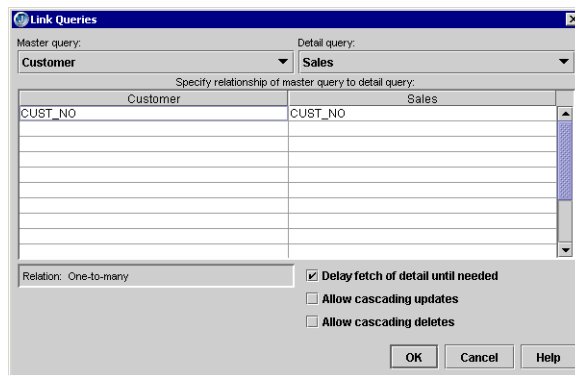
To build multiple queries, choose Queries|Add, and the Data Modeler is ready to begin building a new query. As you select columns in one or more tables, the table names appear in place of the <new query> field.

Specifying a master-detail relationship

To set up a master-detail relationship between two queries,

- 1 Display the Link Queries dialog box in one of two ways:
 - a Choose Queries|Link.
 - b Click-and-drag the mouse pointer in the Queries panel from the query you want to be the master query to the one you want to be the detail query.

Figure 10.7 Link Queries dialog box



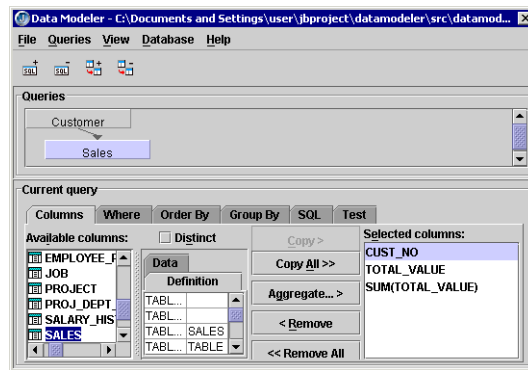
- 2 Select a query to be the master query in the Master Query list.
- 3 Select a query to be the detail query in the Detail Query list.

The Master Query and Detail Query fields are filled with suggested fields. If they are not the ones you want, make the necessary changes.

- 4 Use the table to visually specify the columns that link the master and detail queries together:
 - a Click the first row under the master query column of the table to display a drop-down list of all the specified columns in the master table. Select the column you want the detail data to be grouped under.
 - b Click the first row under the detail query column of the table to display a drop-down list of all columns that are of the same data type and size as the currently selected master column. Select the appropriate column, thereby linking the master and the detail tables together.
 - c Optionally, repeat the preceding two steps for the subsequent rows under the master query column to select additional link columns.
 - d Click OK.

When the Link Queries dialog box closes, an arrow is shown between the two queries in the Queries panel showing the relationship between them.

Figure 10.8 Arrow showing relationship between queries



For more information about master-detail relationships, see [Chapter 9, “Establishing a master-detail relationship.”](#)

Saving your queries

To save the data module you built,

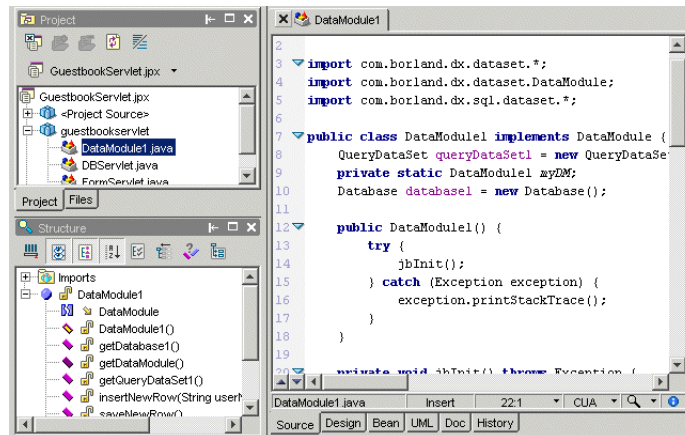
- 1 Choose File|Save in the Data Modeler to save your data module.
- 2 Exit the Data Modeler.

The resulting file appears in your project.

- 3 Compile the data module.

Double-click the file in the project pane to open it in the content pane to view the code the Data Modeler generated.

Figure 10.9 Editor showing code generated by Data Modeler



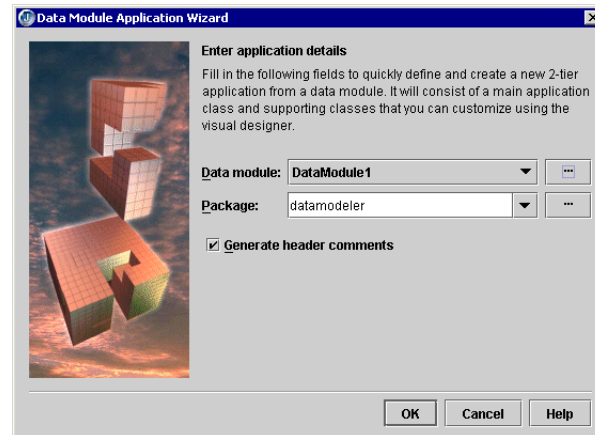
Generating database applications

From your compiled data module, JBuilder can generate two-tier client-server applications with its Data Module Application wizard.

To display the Data Module Application wizard, select the Data Module Application wizard icon in the object gallery:

- 1 Choose File|New and select the General tab.
- 2 Double-click the Data Module Application icon.

Figure 10.10 Data Module Application wizard



- 3 Specify the data module file you want to generate an application from in the dialog box that appears.

You can select any data module that you have or you can select one that was created by the Data Modeler.

- 4 Click OK.

The wizard creates a database application for you. The wizard generates several Java files and an HTML file:

- The files that make up the client are contained in a client two-tier package (the package name is of the form `<project_name>._<data_module>.twotier`):
 - `One or more UIBeans.java` — Each bean implements columnar user interface for a particular `DataSet`.
 - `ClientAboutBoxDialog.java` — Implements the client Help About dialog.
 - `ClientFrame.java` — The client application frame that is the container for the default client user interface. Implements the application menu bar.
 - `ClientResources.java` — Contains client application strings for localization.
- `<datamodule>TwoTierApp.java` — the application
- `<datamodule>AppGenFileList.html` — list of files generated with a brief description of each.

Using a generated data module in your code

Once you've created a data module with the Data Modeler, you can use it in applications that you write. Follow these steps:

- 1 Run the Use DataModule wizard.

In the source code of the frame for your application, it adds a `setModule()` method that identifies the data module. The `setModule()` method the wizard creates calls the frame's `jbInit()` method. The wizard also removes the call to `jbInit()` from the frame's constructor. The frame file must be open in the content pane.

- 2 In the source code of your application file, call the frame's `setModule()` method, passing it the data module class.

For example, suppose you have used the Data Modeler to create a data module called `CountryDataModelModule`. To access the logic stored in that data module in an application you write, you must add a `setModule()` method to your frame class.

To add the `setModule()` method and remove the `jbInit()` method from the frame's constructor,

- 1 Add the data module to the list of required libraries (in Project/Project Properties dialog box).
- 2 Choose Wizards/Edit/Use DataModule while the frame's source code is visible in the editor.
- 3 Specify the data module you want to use with the wizard.
- 4 Select the Application Sets The Instance By Calling `setModule()` option.
- 5 Click OK.

The resulting code of the frame would look like this:

```
package com.borland.samples.dx.myapplication;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
//imports package where data module is
import com.borland.samples.dx.datamodule.*;

public class Frame1 extends JFrame {
    BorderLayout borderLayout1 = new BorderLayout();
    CountryDataModelModule countryDataModelModule1;

    //Construct the frame without calling jbInit()
    public Frame1() {
        enableEvents(AWTEvent.WINDOW_EVENT_MASK);
    }

    //Component initialization
    private void jbInit() throws Exception {
        this.getContentPane().setLayout(borderLayout1);
        this.setSize(new Dimension(400, 300));
        this.setTitle("Frame Title");
    }

    //Overridden so we can exit on System Close
    protected void processWindowEvent(WindowEvent e) {
        super.processWindowEvent(e);
        if(e.getID() == WindowEvent.WINDOW_CLOSING) {
            System.exit(0);
        }
    }
}
```

```

    }

    // The Use DataModule wizard added this code
    public void setModule(CountryDataModelModule countryDataModelModule1) {
        this.countryDataModelModule1 = countryDataModelModule1;
        try {
            jbInit();
        }
        catch(Exception e) {
            e.printStackTrace();
        }
    }
}

```

Note that the frame's `jbInit()` method is now called after the module is set and not in the frame's constructor.

Next you must call the new `setModule()` method from the main source code of your application. In the constructor of the application, call `setModule()`, passing it the data module class. The code of the main application would look like this:

```

package com.borland.samples.dx.myapplication;

import javax.swing.UIManager;

public class Application1 {
    boolean packFrame = false;

    //Construct the application
    public Application1() {
        Frame1 frame = new Frame1();

        // This is the line of code that you add
        frame.setModule(new untitled3.CountryDataModelModule());

        //Validate frames that have preset sizes
        //Pack frames that have useful preferred size info, e.g. from their layout
        if (packFrame)
            frame.pack();
        else
            frame.validate();
        frame.setVisible(true);
    }

    //Main method
    public static void main(String[] args) {
        try {
            UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
        }
        catch(Exception e) {
        }
        new Application1();
    }
}

```


Chapter 11

Filtering, sorting, and locating data

Once you've completed the providing phase of your application and have the data in an appropriate `DataExpress` package `DataSet` component, you're ready to work on the core functionality of your application and its user interface. This chapter demonstrates the typical database application features of filtering, sorting, and locating data.

A design feature of the `DataExpress` package is that the manipulation of data is independent of how the data was obtained. Regardless of which type of `DataSet` component you use to obtain the data, you manipulate it and connect it to controls in exactly the same way. Most of the examples in this chapter use the `QueryDataSet` component, but you can replace this with the `TableDataSet` or any `StorageDataSet` subclass without having to change code in the main body of your application.

Each sample discussed in this chapter is created using the JBuilder IDE and design tools. Wherever possible, we'll use these tools to generate source Java code. Where necessary, we'll show you what code to modify to have your application perform a particular task.

The information presented in this chapter is written with the assumption that you are comfortable using the JBuilder environment. Detailed steps on how to use the user interface are not provided. If you're not yet comfortable with JBuilder, refer to [Chapter 16, "Tutorial: Importing and exporting data from a text file,"](#) "Using JBuilder's IDE" in *Getting Started with JBuilder*, or "Introducing the Designer" in *Designing Applications with JBuilder*.

All of the material in this chapter involves accessing SQL data stored in a local database. For instructions on how to setup and configure JBuilder to use the sample `JDataStore` driver, see ["Adding a JDBC driver to JBuilder" on page 33](#).

We encourage you to use the samples as guides when adding these functions to your application. Finished projects and Java source files, with comments in the source file where appropriate, are provided for many of the concepts presented in this chapter. All files referenced by these examples are found in the JBuilder samples directory.

To create a database application, you first need to connect to a database and provide data to a `DataSet`. ["Retrieving data for the examples" on page 122](#) sets up a query that can be used for some of the examples contained in this chapter. The following list of additional database functionality options (filter, sort, locate data) can be used in any combination, for example, you could choose to temporarily hide all employees whose

last names start with letters between “M” and “Z.” You could sort the employees that are visible by their first names.

- Filtering temporarily hides rows in a `DataSet`.
- Sorting changes the order of a filtered or unfiltered `DataSet`.
- Locating positions the cursor within the filtered or unfiltered `DataSet`.

Retrieving data for the examples

This topic provides the steps for setting up a basic database application that can be used with the examples in this chapter. The query that will be used in these examples is,

```
SELECT * FROM EMPLOYEE
```

This SQL statement selects all columns from a table named `EMPLOYEE`, included in the sample `JDataStore` database (`employee.jds`).

To set up an application for use with the examples,

- 1 Close all open projects (choose `File|Close Project`).
- 2 Choose `File|New Project`.
- 3 Enter a name and location for the project in the Project wizard. Click `Finish`.
- 4 Choose `File|New` from the menu, and double-click the Application icon on the General page of the object gallery.
- 5 Specify the package name and class name in the Application wizard. Click `Finish`.
- 6 Select the `Design` tab to activate the UI designer.
- 7 Click the `Database` component on the `DataExpress` page of the component palette, then click in the component tree or the UI designer to add the component to the application.
- 8 Open the `connection` property editor for the `Database` component by selecting, then clicking the `connection` property ellipsis (...) button in the Inspector. Set the connection properties to the `JDataStore` sample employee table as follows. The Connection URL points to a specific installation location. If you have installed `JBuilder` to a different directory, point to the correct location for your installation.

Property name	Value
Driver	<code>com.borland.datastore.jdbc.DataStoreDriver</code>
URL	Browse to your copy of <code><jbuilder>/samples/JDataStore/datastores/employee.jds</code>
Username	Enter your name (the default is “SYSDBA”)
Password	Enter your password (the default is “masterkey”)

The `connection` dialog box includes a `Test Connection` button. Click this button to check that the connection properties have been correctly set. Results of the connection attempt are displayed in the status area. When the connection is successful, click `OK`. If the connection is not successful, make sure you have followed all the steps for [Chapter 4, “Connecting to a database.”](#)

- 9 Add a `QueryDataSet` component to the designer by clicking the `QueryDataSet` component on the `DataExpress` page and then clicking in the component tree or the UI Designer.

Select the `query` property of the `QueryDataSet` component in the Inspector, click the ellipsis (...) button to open the `QueryDescriptor` dialog box, and set the following properties:

Property name	Value
Database	database1
SQL Statement	SELECT * FROM EMPLOYEE

Click Test Query to ensure that the query is runnable. When the status area indicates Success, click OK to close the dialog box.

- 10 Add a `DBDisposeMonitor` component from the More dbSwing page. The `DBDisposeMonitor` will close the `JDataStore` when the window is closed.
- 11 Set the `dataAwareComponentContainer` property for the `DBDisposeMonitor` to this.

To view the data in your application, add the following UI components and bind them to the data set:

- 1 Select `contentPane(BorderLayout)` in the component tree and set its `layout` property to `null`.
- 2 Drop a `JdbNavToolBar` into the area at the top of the panel in the UI designer. `jdbNavToolBar1` automatically attaches itself to whichever `DataSet` has focus, so you do not need to set its `dataSet` property.
- 3 Drop a `JdbStatusLabel` into the area at the bottom of the panel in the UI designer. `jdbStatusLabel1` automatically attaches itself to whichever `DataSet` has focus, so you do not need to set its `dataSet` property.
- 4 Add a `TableScrollPane` from the dbSwing page to the center of the panel in the UI designer.
- 5 Drop a `JdbTable` into the center of `tableScrollPane1` and set its `dataSet` property to `queryDataSet1`.

You'll notice that the designer displays live data at this point.

Note You may need to position and resize components in the designer to get the application to display properly when it runs.

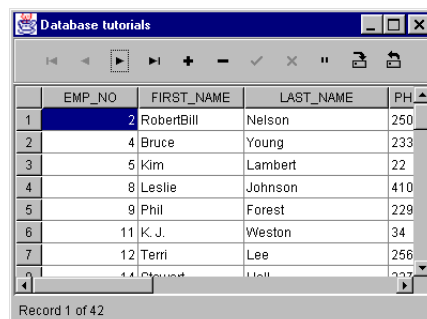
- 6 Choose Run/Run Project to run the application and browse the data set.

The `EMPLOYEE` data set contains 42 records and 11 fields. In the status label for this application, you will see how many records are displaying. When the application is first run, the status label will read `Record 1 of 42`. Some of the examples remove rows from a view. The status label will display the number of rows retrieved into the data set for each application.

For more information on retrieving data for your application, see [Chapter 5, "Retrieving data from a data source."](#)

The running application should look like this:

Figure 11.1 Running database application



Filtering data

Filtering temporarily hides rows in a data set, letting you select, view, and work with a subset of rows in a data set. For example, you may be interested in viewing all orders for a customer, all customers outside the U.S., or all orders that were not shipped within two days. Instead of running a new query each time your criteria change, you can use a filter to show a new view.

In JBuilder, you provide filter code that the data set calls via an event for each row of data to determine whether or not to include each row in the current view. Each time your method is called, it should examine the row passed in, and then indicate whether the row should be included in the view or not. It indicates this by calling `add()` or `ignore()` methods of a passed-in `RowFilterResponse` object.

You hook this code up to the `filterRow` event of a data set using the Events page of the Inspector. When you open the data set, or let it be opened implicitly by running a frame with a control bound to the data set, the filter will be implemented. In this example, we use UI components to let the user request a new filter on the fly.

A filter on a data set is a mechanism for restricting which rows in the data set are visible. The underlying data set is not changed, only the current view of the data is changed and this view is transient. An application can change which records are in the current view “on the fly,” in response to a request from the user (such as is shown in the following example), or according to the application’s logic (for example, displaying all rows to be deleted prior to saving changes to confirm or cancel the operation). When you work with a filtered view of the data and post an edit that is not within the filter specifications, the row disappears from the view, but is still in the data set.

You can work with multiple views of the same data set at the same time, using a `DataSetView`. For more information on working with multiple views of the same data set, see [“Presenting an alternate view of the data” on page 152](#).

Filtering is sometimes confused with sorting and locating.

- Filtering temporarily hides rows in a `DataSet`.
- Sorting changes the order of a filtered or unfiltered `DataSet`. For more information on sorting data, see [“Sorting data” on page 127](#).
- Locating positions the cursor within the filtered or unfiltered `DataSet`. For more information on locating data, see [“Locating data” on page 130](#).

Adding and removing filters

This section describes how to use a data set’s `RowFilterListener` to view only rows that meet the filter criteria. In this example, we create a `JdbTextField` that lets the user specify the column to filter. Then we create another `JdbTextField` that lets the user specify the value that must be in that column in order for the record to be displayed in the view. We add a `JButton` to let the user determine when to apply the filter criteria and show only those rows whose specified column contains exactly the specified value.

In this tutorial, we use a `QueryDataSet` component connected to a `Database` component to fetch data, but filtering can be done on any `DataSet` component.

The finished example is available as a completed project in the `/samples/DataExpress/FilterRows` subdirectory of your JBuilder installation.

To create this application,

- 1 Create a new application by following [“Retrieving data for the examples” on page 122](#). This step enables you to connect to a database, read data from a table, and view and edit that data in a data-aware component.
- 2 Click the Design tab.

- 3 Add two `JdbTextField` components from the `dbSwing` page and a `JButton` component from the `Swing` page. The `JdbTextField` components enable you to enter a field and a value to filter on. The `JButton` component executes the filtering mechanism.
- 4 Define the name of the column to be filtered and its formatter. To do this, select the `Source` tab and add this **import** statement to the existing **import** statements:

```
import com.borland.dx.text.VariantFormatter;
```

- 5 Add these variable definitions to the existing variable definitions in the class definition:

```
Variant v = new Variant();
String columnName = "Last_Name";
String columnValue = "Young";
VariantFormatter formatter;
```

- 6 Specify the filter mechanism.

You restrict the rows included in a view by adding a `RowFilterListener` and using it to define which rows should be shown. The default action in a `RowFilterListener` is to exclude the row. Your code should call the `RowFilterResponse add()` method for every row that should be included in the view. Note that in this example we are checking to see if the `columnName` and `columnValue` fields are blank. If either is blank, all rows are added to the current view.

To create the `RowFilterListener` as an event adapter using the visual design tools,

- a Select the `Design` tab.
- b Select the `queryDataSet1` in the component tree.
- c Select the `Events` tab of the `Inspector`.
- d Select the `filterRow` event.
- e Double-click the `filterRow` value box.

A `RowFilterListener` is automatically generated as an inner class. It calls a new method in your class, called `queryDataSet1_filterRow` method.

- f Add the filtering code to the `queryDataSet1_filterRow` event. You can copy the code from the online help by selecting the code and pressing `Ctrl+C` or selecting `Edit|Copy` from the `Help Viewer` menu.

```
void queryDataSet1_filterRow(ReadRow row, RowFilterResponse
response) {
    try {
        if (formatter == null || columnName == null ||
            columnValue == null || columnName.length() == 0 ||
            columnValue.length() == 0)
            // user set field(s) are blank, so add all rows
            response.add();
        else {
            row.getVariant(columnName, v);
            // fetches row's value of column
            // formats this to a string
            String s = formatter.format(v);
            // true means show this row
            if (columnValue.equals(s))
                response.add();
            else response.ignore();
        }
    }
    catch (Exception e) {
        System.err.println("Filter example failed");
    }
}
```

7 Override the `actionPerformed` event for the `JButton` to retrigger the actual filtering of data. To do this,

- a Select the Design tab.
- b Select the `JButton` in the component tree.
- c Click the Events tab on the Inspector.
- d Select the `actionPerformed` event, and double-click the value box for its event.

The Source tab displays the stub for the `jButton1_actionPerformed` method. The following code uses the adapter class to do the actual filtering of data by detaching and re-attaching the `rowFilterListener` event adapter that was generated in the previous step.

- e Add this code to the generated stub.

```
void jButton1_actionPerformed(ActionEvent e) {

    try {

        // Get new values for variables that the filter uses.
        // Then force the data set to be refiltered.

        columnName = jdbcTextField1.getText();
        columnValue = jdbcTextField2.getText();
        Column column = queryDataSet1.getColumn(columnName);
        formatter = column.getFormatter();

        // Trigger a recalc of the filters

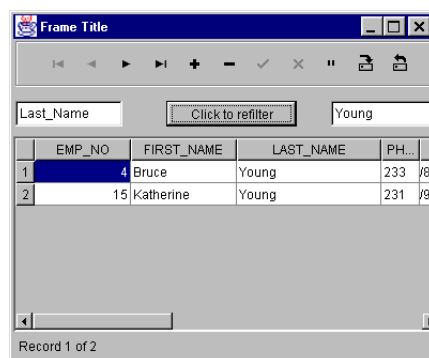
        queryDataSet1.refilter();

        // The table should now repaint only those rows matching
        // these criteria
    }
    catch (Exception ex) {
        System.err.println("Filter example failed");
    }
}
```

8 Compile and run the application.

The running application looks like this:

Figure 11.2 Application running filters



To test this application,

- 1 Enter the name of the column you wish to filter (for example, `Last_Name`) in the first `JdbTextField`.

- 2 Enter the value you wish to filter for in the second `JdbTextField` (for example, Young).
- 3 Click the `JButton`.

Note Leaving either the column name or the value blank removes any filtering and allows all values to be viewed.

Sorting data

Sorting a data set defines an index that allows the data to be displayed in a sorted order without actually reordering the rows in the table on the server.

Data sets can be sorted on one or more columns. When a sort is defined on more than one column, the data set is sorted as follows:

- First on the first column defined in the sort.
- The second column defined in the sort breaks any ties when two or more rows have the same value in the first sort column.
- Subsequent columns defined in the sort continue to break ties.
- If there are still ties after each column defined in the sort has been used, the rows will display in the order they exist in the table on the server.
- If there are still ties after the last column defined in the sort, the columns will display in the order they exist on the table in the server.

You can sort the data in any `DataSet` subclass, including the `QueryDataSet`, `ProcedureDataSet`, `TableDataSet`, and `DataSetView` components. When sorting data in `JBuilder`, note that,

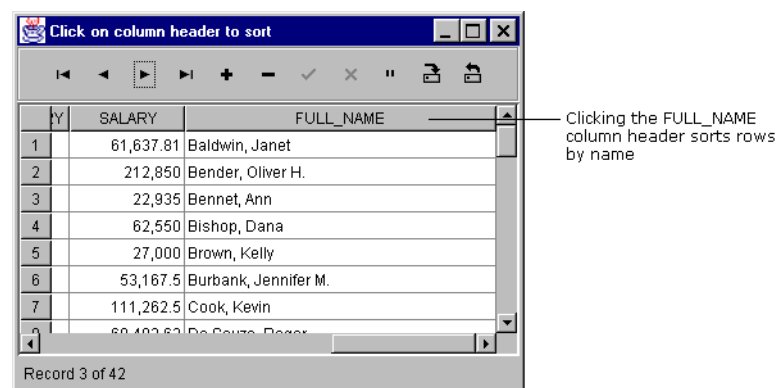
- Case sensitivity applies only when sorting data of type `String`.
- Case sensitivity applies to all `String` columns in a multi-column sort.
- Sort directions (ascending/descending) are set on a column-by-column basis.
- Null values sort to the top in a descending sort, to the bottom in an ascending sort.

Sorting and indexing data are closely related. See [“Understanding sorting and indexing” on page 129](#) for further discussion of indexes.

Sorting data in a `JdbTable`

If your application includes a `JdbTable` that is associated with a `DataSet`, you can sort on a single column in the table by clicking the column header in the running application. Click again to toggle from ascending to descending order.

Figure 11.3 Click on column header to sort at runtime



When sorting data in this way, you can only sort on a single column. Clicking a different column header replaces the current sort with a new sort on the column just selected.

Sorting data using the JBuilder visual design tools

If you need your application to sort in a specified order, the JBuilder visual design tools allow you to quickly set these properties. The `DataSet` `sort` property provides an easy way to,

- View the columns that currently control sort order.
- Select from among the sortable columns in the `DataSet`.
- Add and remove selected columns to and from the sort specification.
- Set the case-sensitivity of the sort.
- Set the sort order to ascending or descending on a column-by-column basis.
- Set unique sort constraints so that only columns with unique key values can be added or updated in a `DataSet`.
- Create a re-usable index for a table.

This example describes how to sort a data set in ascending order by last name. To set sort properties using the JBuilder visual design tools:

- 1 Open or create the project from [“Retrieving data for the examples” on page 122](#).
- 2 Click the Design tab, and select the `QueryDataSet` in the content pane.
- 3 In the Inspector, select, then double-click the area beside the `sort` property.

This displays the `sort` property editor.

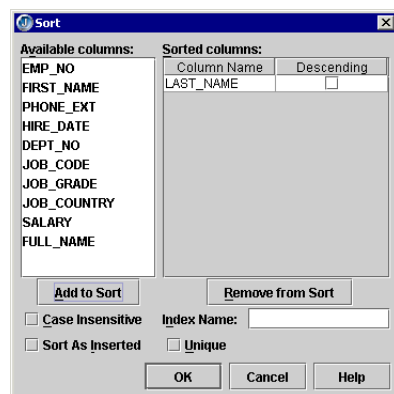
- 4 Specify values for options that affect the sort order of the data.

In this case, select the `LAST_NAME` field from the list of Available Columns, click Add To Sort.

- 5 If you selected the wrong column, click the Remove From Sort button, and redo the previous step.

The dialog box will look like this:

Figure 11.4 Sort property editor



- 6 Click OK.

The property values you specify in this dialog box are stored in a `SortDescriptor` object.

- 7 Choose Run/Run Project to compile and run the application.

It will look like this:

Figure 11.5 Sorted application at runtime

	EMP_NO	FIRST_NAME	LAST_NAME	PH
1	34	Janet	Baldwin	2
2	105	Oliver H.	Bender	255
3	28	Ann	Bennet	5
4	83	Dana	Bishop	290
5	109	Kelly	Brown	202
6	71	Jennifer M.	Burbank	289
7	107	Kevin	Cook	894
8	20	Debra	DeCoursey	200

Understanding sorting and indexing

There are two options on the Sort dialog box that benefit from further discussion: Unique and Index Name. Sorting and indexing are closely related. The following describes the unique option and named indexes in more detail.

Unique

Check the Unique option to create a unique index, which enables a constraint on the data in the `StorageDataSet`. With this constraint, only rows with unique values for the columns defined as `sortKeys` in the `SortDescriptor` can be added or updated in a `DataSet`.

- The Unique option is useful when you're querying data from a server table that has a unique index. Before the user begins editing the data set, you can define a unique index on the columns that are indexed on the server knowing that there will not be any duplicates. This ensures that the user cannot create rows that would be rejected as duplicates when the changes are saved back to the server.
- If a unique index is sorted on more than one column, the constraint applies to all the columns taken together: two rows can have the same value in a single sort column, but no row can have the same value as another row in every sort column.
- Unique is a constraint on the data set, not just on the index. If you define a unique index on a column, you are asserting that no two rows in the data set have the same value in that column. If there are two or more rows in the data set that have the same value in the unique column *when the index is first created*, rows that violate the unique constraint are copied into a separate `DataSet`. You can access this `DataSet` by calling the `StorageDataSet.getDuplicates()` method. The duplicates `DataSet` can be deleted by calling the `StorageDataSet.deleteDuplicates()` method.
- You can have one or more unique `sort` property settings for a `StorageDataSet` at one time. If a duplicates `DataSet` exists from a previous unique `sort` property setting, additional unique `sort` property settings cannot be made until the earlier duplicates have been deleted. This is done to protect you from eliminating valuable rows due to an erroneous unique `sort` property setting.

Index Name

Enter a name in the Index Name field to create a named index. This is the user-specified name to be associated with the sort specification (index) being defined in the Sort property editor. The intent of the named index is to let you revert to a previously defined sort.

The index has been *maintained*, so that it can be reused. That is, each index is updated (kept current) to reflect insertions, deletions, and edits to its sort column or columns. For example, if you define a unique sort on the CustNo column of your Customers data set, then decide you want to see customers by ZIP code and define a sort to show that, you still can't enter a new customer with a duplicate CustNo value.

The named index implements the sort orders (that is, indexes), and whether or not the unique constraint is enforced, under an easy-to-retrieve setting, even if you stop viewing data in that order.

To view a data set in the order defined by an existing named index, set its `sort` property using the `sortDescriptor` constructor that takes just an index name. If you set a data set's `sort` property to a new `sortDescriptor` with exactly the same parameters as an existing sort, the existing sort is used.

Sorting data in code

You can enter the code manually or use JBuilder design tools to generate the code for you to instantiate a `SortDescriptor`. The code generated automatically by the JBuilder design tools looks like this:

```
queryDataSet1.setSort(new com.borland.dx.dataset.SortDescriptor("",
    new String[] { "LAST_NAME", "FIRST_NAME", "EMP_NO" }, new boolean[]
    { false, false, false, }, true, false, null));
```

In this code segment, the `sortDescriptor` is instantiated with sort column of the last and first names fields (`LAST_NAME` and `FIRST_NAME`), then the employee number field (`EMP_NO`) is used as a tie-breaker in the event two employees have the same name. The sort is case insensitive, and in ascending order.

To revert to a view of unsorted data, close the data set, and set the `setSort` method to null, as follows. The data will then be displayed in the order in which it was added to the table.

```
queryDataSet1.setSort(null);
```

Locating data

A basic need of data applications is to find specified data. This topic discusses the following two types of locates:

- An interactive locate using a `JdbNavField`, where the user can enter values to locate when the application is running.
- A locate where the search values are programmatically set.

Locating data with a JdbNavField

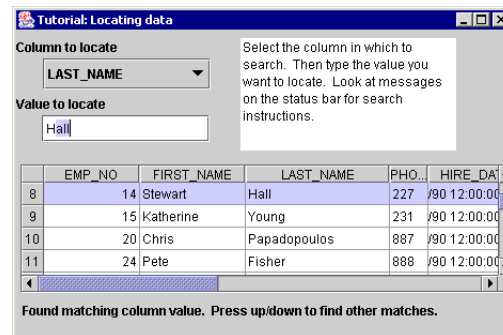
The `dbSwing` library includes a `JdbNavField` component that provides locate functionality in a user-interface control. The `JdbNavField` includes an incremental search feature for `String` type columns. Its `columnName` property specifies the column in which to perform the locate. If not set, the locate is performed on the last column visited in the `JdbTable`.

If you include a `JdbStatusLabel` component in your application, `JdbNavField` prompts and messages are displayed on the status label.

The `/samples/DataExpress/LocatingData` directory in your JBuilder installation includes a finished example of an application that uses the `JdbNavField` under the project name `LocatingData.jpx`. This sample shows how to set a particular column for the locate

operation as well as using a `JdbComboBox` component to enable the user to select the column in which to locate the value. The completed application looks like this:

Figure 11.6 Sample application with `JdbNavField`



To create this application,

- 1 Create a new application by following “Retrieving data for the examples” on page 122.

This step enables you to connect to a database, read data from a table, and view and edit that data in a data-aware component. Check the preceding screen shot of the running application for the approximate positioning of components.

- 2 Add a `JdbNavField` from the More dbSwing page of the component palette to the UI designer, and set its `dataSet` property to `queryDataSet1`.
- 3 Add a `JdbComboBox` from the dbSwing page of the component palette to the UI designer.
- 4 Set the `items` property for `jdbComboBox1` to the column name values `EMP_NO`, `FIRST_NAME`, and `LAST_NAME`.
- 5 Select the Events tab of the Inspector. Select the `itemStateChanged()` event for `jdbComboBox1`, and double-click its value field. A stub for the `itemStateChanged()` event is added to the source, and the cursor is positioned for insertion of the following code, which allows the user to specify the column in which to locate data.

```
void jdbComboBox1_itemStateChanged(ItemEvent e) {
    jdbNavField1.setColumnName(jdbComboBox1.getSelectedItem().toString());
    jdbNavField1.requestFocus();
}
```

This code tests for the a change in the `JdbComboBox`. If it determines that a different column value is selected, the `columnName` property for the `JdbNavField` is set to the column named in the `JdbComboBox`. This instructs the `JdbNavField` to perform locates in the specified `Column`. Focus is then shifted to the `JdbNavField` so that you can enter the value to search for.

- 6 Add a `JdbTextArea` component from the dbSwing page.

Place the `JdbTextArea` component next to the `JdbComboBox` component in the UI designer. Set its `text` property so that the user knows to select a column on which to locate data, for example,

```
Select the column in which to search. Then type the value you want to locate.
Look at messages on the status bar for search instructions.
```

Alternatively, if you want to locate only in a particular `Column`, you could set the `JdbNavField` component’s `columnName` property to the `DataSet` column on which you want to locate data, for example, `LAST_NAME`.

- 7 Add a `JdbLabel` from the `dbSwing` page. Place it next to `jdbNavField1`. Set its `text` property to: Value to locate.

Note See the screen shot of the running application earlier in this section for additional instructional text.

- 8 Run the application.

The status label updates to reflect the current status of the application. For example,

- Select the column name on which you want to perform the locate in the `JdbComboBox`. The status area displays:

Enter a value and press enter to begin search.

- Start typing the value to locate in the `JdbNavField`. If you're locating in a `String` column, as you type, notice that the `JdbNavField` does an incremental search on each key pressed. For all other data types, press *Enter* to perform the locate. If a value is not found in the table, the status area displays:

Could not find a matching column value.

- Press the *Up Arrow* or *Down Arrow* keys to perform a "locate prior" or "locate next" respectively. When a matching value is found, the status area displays:

Found matching column value. Press up/down to find other matches.

Locating data programmatically

This section explores the basics of locating data programmatically as well as conditions which affect the locate operation.

When programmatically locating data,

- 1 Instantiate a `DataRow` based on the `DataSet` you want to search. If you don't want to search on all columns in the `DataSet`, create a "scoped" `DataRow` (a `DataRow` that contains just the columns for which you want to specify locate values). (See ["Locating data using a DataRow" on page 133.](#))
- 2 Assign the values to locate in the appropriate columns of the `DataRow`.
- 3 Call the `locate(ReadRow, int)` method, specifying the location options you want as the `int` parameter. Test the return value to determine if the locate succeeded or failed.
- 4 To find additional occurrences, call `locate()` again, specifying a different locate option, for example, `Locate.NEXT` or `Locate.LAST`. See the `Locate` class Field Summary for information on all the `Locate` options.

The core locate functionality uses the `locate(ReadRow, int)` method. The first parameter, `ReadRow`, is of an abstract class type. Normally you use its (instantiable) subclass `DataRow` class. The second parameter represents the locate option and is defined in the Field Summary for the `Locate` class. The `Locate` class fields represent options that let you control where the search starts from and how it searches, for example with or without case sensitivity. (For more information on locate options, see ["Working with locate options" on page 133.](#)) If a match is found, the current row position moves to that row. All data-aware components that are connected to the same located `DataSet` navigate together to the located row.

The `Locate()` method searches within the current view of the `DataSet`. This means that rows excluded from display by a `RowFilterListener` are not included in the search.

The view of the `DataSet` can be sorted or unsorted; if it is sorted, the `locate()` method finds matching rows according to the sort sequence.

To locate a `null` value in a given column of a `DataSet`, include the column in the `DataRow` parameter of the `locate()` method but do not assign it a value.

Tip If the `locate()` method fails to find a match when you think it should succeed, check for null values in some columns; remember that all columns of the `DataRow` are included in the search. To prevent this, use a “scoped” `DataRow` containing only the desired columns.

Locating data using a DataRow

A `DataRow` is similar to a `DataSet` in that it contains multiple `Column` components. However, it stores only one row of data. You specify the values to locate for in the `DataRow`.

When the `DataRow` is created based on the same located `DataSet`, the `DataRow` contains the same column names and data types and column order as the `DataSet` it is based on. All columns of the `DataRow` are included in the locate operation by default; to exclude columns from the locate, create a *scoped* `DataRow` that contains only specified columns from the `DataSet`. You create a scoped `DataRow` using either of the following `DataRow` constructors:

- `DataRow(DataSet, String)`
- `DataRow(DataSet, String[])`

Both the `DataRow` and the `DataSet` are subclasses of `ReadWriteRow`. Both inherit the same methods for manipulation of its contents, for example, `getInt(String)`, and `setInt(String, int)`. You can therefore work with `DataRow` objects using many of the same methods as the `DataSet`.

Working with locate options

You control the locate operation using locate options. These are constants defined in the `com.borland.dx.dataset.Locate` class. You can combine locate options using the bitwise OR operator; several of the most useful combinations are already defined as constants. Four of the locate options (`FIRST`, `NEXT`, `LAST`, and `PRIOR`) determine how the rows of the `DataSet` are searched. The `CASE_INSENSITIVE` and `PARTIAL` options define what is considered a matching value. The `FAST` constant affects the preparation of the locate operation.

You must specify where the locate starts searching and which direction it moves through the rows of the `DataSet`. Choose one of the following options:

- `FIRST` starts at the first row, regardless of your current position, and moves down.
- `LAST` starts at the last row and moves up.
- `NEXT` starts at your current position and moves down.
- `PRIOR` starts at your current position and moves up.

If one of these constants is not specified for a locate operation, a `DataSetException` of `NEED_LOCATE_START_OPTION` is thrown.

To find all matching rows in a `DataSet`, call the `locate()` method once with the locate option of `FIRST`. If a match is found, re-execute the locate using the `NEXT_FAST` option, calling the method with this locate option repeatedly until it returns `false`. The `FAST` locate option specifies that the locate values have not changed, so they don't need to be read from the `DataRow` again. To find all matching rows starting at the bottom of the view, use the options `LAST` and `PRIOR_FAST` instead.

The `CASE_INSENSITIVE` option specifies that string values are considered to match even if they differ in case. Specifying whether a locate operation is `CASE_INSENSITIVE` or not is optional and only has meaning when locating in `String` columns; it is ignored for other data types. If this option is used in a multi-column locate, the case sensitivity applies to all `String` columns involved in the search.

The `PARTIAL` option specifies that a row value is considered to match the corresponding locate value if it starts with the first characters of the locate value. For example, you

might use a locate value of “M” to find all last names that start with “M”. As with the `CASE_INSENSITIVE` option, `PARTIAL` is optional and only has meaning when searching `String` columns.

Multi-column locates that use `PARTIAL` differ from other multi-column locates in that the order of the locate columns makes a difference. The constructor for a scoped, multi-column `DataRow` takes an array of column names. These names need not be listed in the order that they appear in the `DataSet`. The `PARTIAL` option applies only to the last column specified, therefore, control over which column appears last in the array is important.

For a multi-column locate operation using the `PARTIAL` option to succeed, a row of the `DataSet` must match corresponding values for all columns of the `DataRow` except the last column of the `DataRow`. If the last column starts with the locate value, the method succeeds. If not, the method fails. If the last column in the `DataRow` is not a `String` column, the `locate()` method throws a `DataSetException` of `PARTIAL_SEARCH_FOR_STRING`.

Locates that handle any data type

Data stored in DataExpress components are stored in `Variant` objects. When data is displayed, a `String` representation of the variant is used. To write code that performs a generalized locate that handles columns of any data type, use one of the `setVariant()` methods and one of the `getVariant()` methods.

For example, you might want to write a generalized locate routine that accepts a value and looks for the row in the `DataSet` that contains that value. The same block of code can be made to work for any data type because the data stays a variant. To display the data, use the appropriate formatter class or create your own custom formatter.

Column order in the DataRow and DataSet

While a `Column` from the `DataSet` can only appear once in the `DataRow`, the column order may be different in a scoped `DataRow` than in the `DataSet`. For some locate operations, column order can make a difference. For example, this can affect multi-column locates when the `PARTIAL` option is used. For more information on this, see the paragraph on multi-column locates with the `PARTIAL` option, on [page 134](#).

Chapter 12

Adding functionality to database applications

Once you've completed the providing phase of your application and have the data in an appropriate `DataExpress` package `DataSet` component, you're ready to work on the core functionality of your application and its user interface. [Chapter 11, "Filtering, sorting, and locating data"](#) introduced sorting, filtering, and locating data in a data set. This chapter demonstrates other typical database applications.

A design feature of the `DataExpress` package is that the manipulation of data is independent of how the data was obtained. Regardless of which type of `DataSet` component you use to obtain the data, you manipulate it and connect it to controls in exactly the same way. Most of the examples in this chapter use the `QueryDataSet` component, but you can replace this with the `TableDataSet` or any `StorageDataSet` subclass without having to change code in the main body of your application.

Each sample is created using the JBuilder IDE and design tools. Wherever possible, we'll use these tools to generate Java source code. Where necessary, we'll show you what code to modify, where, and how, to have your application perform a particular task.

The examples and tutorials referenced in this chapter involve accessing SQL data stored in a local `JDataStore`. Finished projects and Java source files are provided in the JBuilder `DataExpress` samples directory (`/samples/DataExpress`), with comments in the source file where appropriate. We encourage you to use the samples as guides when adding these functions to your application.

To create a database application, you first need to connect to a database and provide data to a `DataSet`. The steps for connecting to a database are described in [Chapter 17, "Tutorial: Creating a basic database application."](#) The tutorial sets up a query that can be used for each of the following database tasks:

- "Using pick lists and lookups"
- "Using calculated columns"
- "Aggregating data with calculated fields"
- "Adding an Edit or Display Pattern for data formatting"
- "Presenting an alternate view of the data"
- "Ensuring data persistence"
- "Using variant data types"

Using pick lists and lookups

A `Column` can derive its values in the following ways:

- From data in a database column
- As a result of being imported from a text file
- As a result of a calculation, which can include calculated columns, aggregated data, data looked up in another data set, or data that is chosen via a pick list.

This section covers providing values to a column using a pick list to enter a new value to a column, and it also covers creating a lookup that will display values from another column.

Note Pick lists are used for *editing* a dataset. That is, by selecting a value from a pick list, the user identifies the row in the pick list dataset that will provide data for the row being edited in the target dataset. Lookups, on the other hand, are used for *displaying* information. A lookup is used to display a more useful interpretation of data, but the actual data in the target dataset is not changed.

Data entry with a pick list

Pick lists can significantly speed data entry. A pick list can be used to select a value from a column of another dataset (the pick list dataset) for data entry in the target dataset. Using a pick list, the available choices can be displayed in a drop-down list. `PickListDescriptor` parameters specify which columns in the pick list dataset are displayed in the drop-down list and how values will be copied from the pick list dataset to the current row of the target dataset.

The following section includes describes how to create (using JBuilder design tools) a pick list that can be used to set the value of a column from the list of values available in another data set. The instructions include steps for looking up a value in a pick list for data entry purposes, in this case for selecting a country for a customer or employee. In this example, the `pickList` property of a column allows you to define which column of which data set will be used to provide values for the pick list. The choices will be available for data entry in a visual component, such as a table, when the application is running.

When the application is running, you can insert a row into the table, and, when it you enter a value for the `JOB_COUNTRY` field, you can select it from the drop-down pick list. The country you select is automatically inserted into the `JOB_COUNTRY` field in the `EMPLOYEE` data set.

Adding a pick list field

The following steps show how to create a pick list that can be used to set the value of the `JOB_COUNTRY` column from the list of countries available in the `COUNTRY` table. When the user selects a country from the picklist, that selection is automatically written into the current field of the table. The sample project, `Picklist.jpx`, located in the `/samples/DataExpress/Picklist` subdirectory of your JBuilder installation, is a completed application that uses the pick list described in the following steps.

- 1 Create a simple database application, as described in [“Retrieving data for the examples” on page 122](#).

- 2 Add another `QueryDataSet` to the application.

This will form the query to populate the list of choices.

- 3 Select the `query` property of `queryDataSet2` in the Inspector, click the ellipsis (...) button to open the Query property editor, and set the `query` property as follows:

Option	Value
Database	database1
SQL Statement	select COUNTRY from COUNTRY

- 4 Click Test Query, and when successful, click OK to close the dialog box.
- 5 Expand the `queryDataSet1` component in the component tree to expose all of the columns, and select `JOB_COUNTRY`.
- 6 Select the `pickList` property in the Inspector, click the ellipsis (...) button to open the PickList property editor, and set the `pickList` properties as follows:

Property name	Value
Picklist/Lookup DataSet	<code>queryDataSet2</code>
<code>queryDataSet2</code>	<code>COUNTRY</code>
Data Type	<code>STRING</code>
Show In Picklist	<code>checked</code>
<code>queryDataSet1</code>	<code>JOB_COUNTRY</code>

Click OK.

- 7 Click the Source tab, and enter the following code after the call to `jbInit()`:

```
queryDataSet2.open();
```

This opens `queryDataSet2`, which is attached to the `EMPLOYEE_PROJECT` table. Normally, a visual, data-aware component such as `JdbTable` would open the data set for you automatically, but in this case, there is no visual component attached to this data set, so it must be opened explicitly.

- 8 Run the application by choosing Run!Run Project.

When the application is running, you can insert a row into the table, and, when it you enter a value for the `JOB_COUNTRY` field, you can select it from the drop-down pick list. The country you select is automatically inserted into the `JOB_COUNTRY` field in the `EMPLOYEE` data set.

Removing a pick list field

To remove a pick list,

- 1 Select the column that contains the pick list in the component tree.
- 2 Open the PickList property editor by clicking the ellipsis (...) button in the `pickList` property in the Inspector.
- 3 Set the PickList/Lookup DataSet field to `<none>` and click OK.
- 4 Alternatively, a pick list can be removed by right-clicking the property in the Inspector, and choosing Clear Property Setting from the context menu.

Create a lookup using a calculated column

This type of lookup retrieves values from a specified table based on criteria you specify and displays it as part of the current table. To create a calculated column, you need to create a new `Column` object in the `StorageDataSet`, set its `calcType` appropriately, and code the `calcFields` event handler. The lookup values are only visible in the running application. Lookup columns can be defined and viewed in JBuilder, but JBuilder-defined lookup columns exist in the JBuilder dataset, not in the table in the database that provides the rest of the dataset's data. These lookup columns can, however, be exported to a text file.

An example of looking up a field in a different table for display purposes is looking up a part number to get a part description for display in an invoice line item or looking up a postal code for a specified city and state.

The `lookup()` method uses specified search criteria to search for the first row matching the criteria. When the row is located, the data is returned from that row, but the cursor is not moved to that row. The `locate()` method is a method that is similar to `lookup()`, but actually moves the cursor to the first row that matches the specified set of criteria. For more information on the `locate()` method, see [“Locating data” on page 130](#).

The `lookup()` method can use a scoped `DataRow` (a `DataRow` with less columns than the `DataSet`) to hold the values to search for and options defined in the `Locate` class to control searching. This scoped `DataRow` will contain only the columns that are being looked up and the data that matches the current search criteria, if any. With lookup, you generally look up values in another table, so you will need to instantiate a connection to that table in your application.

This example shows how to use a calculated column to search and retrieve an employee name (from `EMPLOYEE`) for a given employee number in `EMPLOYEE_PROJECT`. This type of lookup field is for display purposes only. The data this column contains at run time is not retained because it already exists elsewhere in your database. The physical structure of the table and data underlying the data set is not changed in any way. The lookup column will be read-only by default. This project can be viewed as a completed application by running the sample project `Lookup.jpx`, located in the `/samples/DataExpress/Lookup` directory of your JBuilder installation.

For more information on using the `calcFields` event to define a calculated column, refer to [“Using calculated columns” on page 142](#).

- 1 Create a new application by following [“Retrieving data for the examples” on page 122](#).

This step enables you to connect to a database, read data from a table, and view and edit that data in a data-aware component.

- 2 Add another `QueryDataSet` to the application.

This will provide data to populate the base table where we later add columns to perform lookups to other tables. Set the `query` property of `queryDataSet2` as follows:

For this option	Make this choice
Database	<code>database1</code>
SQL Statement	<code>select * from EMPLOYEE_PROJECT</code>

- 3 Click Test Query, and when successful, click OK to close the dialog box.
- 4 Select the `JdbTable` in the content pane, and change its `dataSet` property to `queryDataSet2`.

This will enable you to view data in the designer and in the running application.

- Click the expand icon to the left of the `queryDataSet2` in the component tree to expose all of the columns, select `<new column>`, and set the following properties in the Inspector for the new column:

Property name	Value
<code>calcType</code>	CALC
<code>caption</code>	EMPLOYEE_NAME
<code>columnName</code>	EMPLOYEE_NAME
<code>dataType</code>	STRING

The new column will display in the list of columns and in the table control. You can adjust the column display position for this or any column with the `preferredOrdinal` property. No data will be displayed in the lookup column in the table in the designer. The lookups are only visible when the application is running. The data type of `STRING` is used here because that is the data type of the `LAST_NAME` column which is specified later as the lookup column. Calculated columns are read-only, by default.

- Select the Events tab of the Inspector (assuming the new column is still selected in the content pane), and select, then double-click the `calcFields` event.

The cursor is positioned in the appropriate location in the Source pane.

- Enter the following code, which actually performs the lookup and places the looked-up value into the newly-defined column.

```
void queryDataSet2_calcFields(ReadRow changedRow, DataRow
    calcRow, boolean isPosted) throws DataSetException{
    // Define a DataRow to hold the employee number to look for
    // in queryDataSet1, and another to hold the row of employee
    // data that we find.
    DataRow lookupRow = new DataRow(queryDataSet1, "EMP_NO");
    DataRow resultRow = new DataRow(queryDataSet1);

    // The EMP_NO from the current row of queryDataSet2 is our
    // lookup criteria.
    // We look for the first match, since EMP_NO is unique.
    // If the lookup succeeds, concatenate the name fields from
    // the employee data, and put the result in dataRow;
    // otherwise, let the column remain blank.

    lookupRow.setShort("EMP_NO", changedRow.getShort("EMP_NO"));
    if (queryDataSet1.lookup(lookupRow, resultRow,
        Locate.FIRST))
        calcRow.setString("EMPLOYEE_NAME",
            resultRow.getString("FIRST_NAME") +
            " " + resultRow.getString("LAST_NAME"));
    }
}
```

- Click the Source tab, and enter the following code after the call to `jbInit()`.

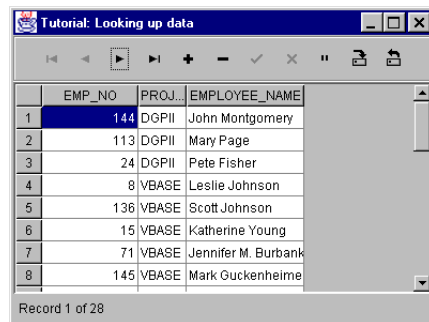
```
queryDataSet1.open();
```

This opens `queryDataSet1`, which is attached to the `EMPLOYEE` table. Normally, a visual, data-aware component such as `JdbTable` would open the data set for you automatically, but in this case, there is no visual component attached to this data set, so it must be opened explicitly.

- Choose Run|Run Project to run the application.

The running application will look like this:

Figure 12.1 Lookup application



When the application is running, the values in the calculated lookup column will automatically adjust to changes in any columns, in this case the EMP_NO column, referenced in the calculated values. If the EMP_NO field is changed, the lookup will display the value associated with the current value when that value is posted.

Create a lookup using the PickListDescriptor parameters

A lookup can be created using the `setPickList` method of the `Column` class. This method specifies the `PickListDescriptor` for the `Column`, which describes the relationship between the `Column` and a second, separate “pick list” or “lookup” `DataSet`. The `PickList` property editor, lets you edit the `PickListDescriptor` parameters. To open the `PickList` property editor, select the `pickList` property of a `Column` in the Inspector, and click the ellipsis (...) button.

The `PickList` property editor specifies the following properties for a lookup:

Property	Description
<code>pickListDataSet</code>	Set with the Picklist/Lookup Dataset drop-down list box, this property specifies the source (lookup) <code>DataSet</code> that contains the items to look up and display in the target <code>DataSet</code> .
<code>pickListColumns</code>	This value corresponds to the value in the first column of the table in the row for which the <code>destinationColumns</code> property was set, and specifies the columns of the <code>pickListDataSet</code> from which values in the selected row are copied to (or displayed instead of) <code>destinationColumns</code> .
<code>pickListDisplayColumns</code>	Set with the Show In Picklist check box, this property specifies the <code>Column</code> components of the <code>DataSet</code> to display in the pick list. This property is not used for lookups, so the check box is unchecked.
<code>destinationColumns</code>	This value is set in the last column of the table shown in the <code>PickList</code> property editor, and specifies the <code>Column</code> components of the target <code>DataSet</code> that are populated with the values associated with the selected pick list/lookup choice.
<code>lookupDisplayColumn</code>	Set with the Lookup Column To Display drop-down list box, this property specifies which column to display (field display values) when the source data is not open. This property is used when the displayed items list differs from the values stored when an item is selected.
<code>enforceIntegrity</code>	Set with the Enforce Integrity check box. This property determines whether data integrity rules are enforced on the data added to <code>destinationColumns</code> . This property is not currently used.

Example

The sample project, `Picklist.jpx`, located in the `/samples/DataExpress/Picklist` subdirectory of your JBuilder installation, is a completed application that uses a pick list. You can modify the application as described in the following steps to perform a lookup to display a complete job title instead of a job code for each employee.

To define a lookup with the `PickList` property editor,

- 1 Add another `QueryDataSet` to the application.

This new `QueryDataSet`, `queryDataSet3`, will be used as the lookup `DataSet`.

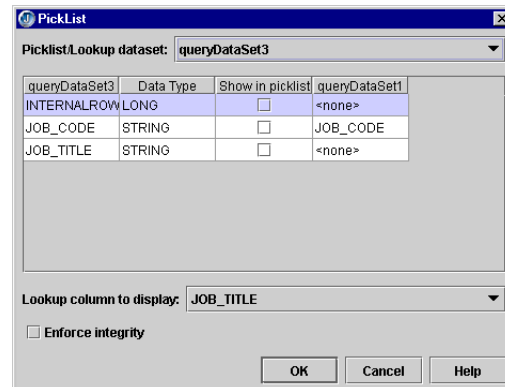
- 2 Select the `query` property of `queryDataSet2` in the Inspector, click the ellipsis (...) button to open the Query property editor, and set the `query` property as follows:

Option	Value
Database	database1
SQL Statement	SELECT JOB.JOB_CODE, JOB.JOB_TITLE FROM JOB

- 3 In the structure pane, select the `JOB_CODE` column in `queryDataSet1`, select the `pickList` property in the Inspector, and click the ellipsis (...) button to open the `PickList` property editor.
- 4 Set the following properties in the `PickList` property editor, and click OK.

Property name	Value
Picklist/Lookup DataSet	queryDataSet3
queryDataSet3	JOB_CODE
Data Type	STRING
Show In Picklist	Unchecked
queryDataSet1	JOB_CODE

The property editor should look like the following image:



- 5 Click the Source tab, and look at the `PickListDescriptor` entry that was generated.

```
column3.setPickList(new com.borland.dx.dataset.PickListDescriptor(queryDataSet3,
    new String[] {"JOB_CODE"}, null, new String[] {"JOB_CODE"}, "JOB_TITLE", false));
```

As illustrated in the code, the `PickListDescriptor` properties are set to the following values:

Property	Value
<code>pickListDataSet</code>	<code>queryDataSet3</code>
<code>pickListColumns</code>	<code>new String[] {"JOB_CODE"} (from the Job table in the lookup DataSet)</code>

Property	Value
<code>pickListDisplayColumns</code>	<code>null</code>
<code>destinationColumns</code>	<code>new String[] { "JOB_CODE" }</code> (in the Employee table in the target DataSet)
<code>lookupDisplayColumn</code>	<code>"JOB_TITLE"</code>
<code>enforceIntegrity</code>	<code>false</code>

6 Run the application by choosing Run/Run Project.

The JOB_CODE column should now be replaced by a JOB_TITLE column, which displays complete job titles looked up from `queryDataSet3`.

Using calculated columns

Typically, a `Column` in a `StorageDataSet` derives its values from data in a database column or as a result of being imported from a text file. A column may also derive its values as a result of a calculated expression. JBuilder supports two kinds of calculated columns: calculated and aggregated.

To create a calculated column, you need to create a new persistent `Column` object in the `StorageDataSet` and supply the expression to the `StorageDataSet` object's `calcFields` event handler. Calculated columns can be defined and viewed in JBuilder. The calculated values are only visible in the running application. JBuilder-defined calculated columns exist in the JBuilder dataset, not in the table in the database that provides the rest of the dataset's data. These calculated columns can, however, be written to a text file. For more information on defining a calculated column in the designer, see [“Create a calculated column in the designer” on page 143](#). For more information on working with columns, see [Chapter 7, “Working with columns.”](#)

The formula for a calculated column generally uses expressions involving other columns in the data set to generate a value for each row of the data set. For example, a data set might have non-calculated columns for QUANTITY and UNIT_PRICE and a calculated column for EXTENDED_PRICE. EXTENDED_PRICE would be calculated by multiplying the values of QUANTITY and UNIT_PRICE.

Calculated aggregated columns are used to group and summarize data, for example, to summarize total sales by quarter. Aggregation calculations can be specified completely through property settings and any number of columns can be included in the grouping. Four types of aggregation are supported (sum, count, min, and max) as well as a mechanism for creating custom aggregation methods. For more information, see [“Aggregating data with calculated fields” on page 144](#).

Calculated columns are also useful for holding lookups from other tables. For example, a part number can be used to retrieve a part description for display in an invoice line item. For information on using a calculated field as a lookup field, see [“Using pick lists and lookups” on page 136](#).

Values for all calculated columns in a row are computed in the same event call.

These are the topics covered:

- [“Create a calculated column in the designer” on page 143](#)
- [“Aggregating data with calculated fields” on page 144](#)
- [“Setting properties in the AggDescriptor” on page 147](#)
- [“Creating a custom aggregation event handler” on page 147](#)

Create a calculated column in the designer

This example builds on the example in [“Retrieving data for the examples” on page 122](#). The database table that is queried is `EMPLOYEE`. The premise for this example is that the company is giving all employees a 10% raise. We create a new column named `NEW_SALARY` and create an expression that multiplies the existing `SALARY` data by 1.10 and places the resulting value in the `NEW_SALARY` column. The completed project is available in the `/samples/DataExpress/CalculatedColumn` directory of your JBuilder installation under the project name `CalculatedColumn.jpx`.

- 1 Create a new application by following [“Retrieving data for the examples” on page 122](#).

This step enables you to connect to a database, read data from a table, and view and edit that data in a data-aware component.

- 2 Click the expand icon beside `queryDataSet1` in the component tree to display all columns, select `<new column>`, and set the following properties in the Inspector:

Property name	Value
<code>calcType</code>	<code>CALC</code>
<code>caption</code>	<code>NEW_SALARY</code>
<code>columnName</code>	<code>NEW_SALARY</code>
<code>dataType</code>	<code>BIGDECIMAL</code>
<code>currency</code>	<code>true</code>

If you were adding more than one column, you could adjust the column display position for this or any other persistent column with the `preferredOrdinal` property. No data will be displayed in the calculated column in the table in the designer. The calculations are only visible when the application is running. The data type of `BIGDECIMAL` is used here because that is the data type of the `SALARY` column which will be used in the calculation expression. Calculated columns are always read-only.

- 3 Select the `queryDataSet1` object, select the Events tab of the Inspector, select the `calcFields` event handler, and double-click its value.

This creates the stub for the event’s method in the Source window.

- 4 Modify the event method to calculate the salary increase, as follows:

```
void queryDataSet1_calcFields(ReadRow changedRow, DataRow
    calcRow, boolean isPosted) throws DataSetException{
    //calculate the new salary
    calcRow.setBigDecimal("NEW_SALARY",
        changedRow.getBigDecimal("SALARY").multiply(new
        BigDecimal(1.1)));
}
```

This method is called for `calcFields` whenever a field value is saved and whenever a row is posted. This event passes in an input which is the current values in the row (`changedRow`), an output row for putting any changes you want to make to the row (`calcRow`), and a boolean (`isPosted`) that indicates whether the row is posted in the `DataSet` or not. You may not want to recalculate fields on rows that are not posted yet.

- 5 Import the `java.math.BigDecimal` class to use a `BIGDECIMAL` data type. Add this statement to the existing `import` statements.

```
import java.math.BigDecimal;
```

- 6 Run the application to view the resulting calculation expression.

When the application is running, the values in the calculated column will automatically adjust to changes in any columns referenced in the calculated expression. The `NEW_SALARY` column displays the value of $(\text{SALARY} * 1.10)$. The running application looks like this:

Figure 12.2 Calculated columns

	FULL_NAME	SALARY	New_SALARY
1	Baldwin, Janet	61,637.81	67,801.59
2	Bender, Oliver H.	212,850	234,135.00
3	Bennet, Ann	22,935	25,228.50
4	Bishop, Dana	62,550	68,805.00
5	Brown, Kelly	27,000	29,700.00
6	Burbank, Jennifer M.	53,167.5	58,484.25
7	Cook, Kevin	111,262.5	122,388.75
8	De Silva, Dany	60,402.63	66,442.89

Record 1 of 42

Aggregating data with calculated fields

You can use the aggregation feature of a calculated column to summarize your data in a variety of ways. Columns with a `calcType` of `aggregated` have the ability to

- Group and summarize data to determine bounds.
- Calculate a sum.
- Count the number of occurrences of a field value.
- Define a custom aggregator using your own method of aggregation.

The `AggDescriptor` is used to specify columns to group, the column to aggregate, and the aggregation operation to perform. The `aggDescriptor` is described in more detail in the following sections. The aggregation operation is an instance of one of these classes:

- `CountAggOperator`
- `SumAggOperator`
- `MaxAggOperator`
- `MinAggOperator`
- A custom aggregation class defined by you

Creating a calculated aggregated column is simpler than creating a calculated column, because no event method is necessary (unless you are creating a custom aggregation component). The aggregate can be computed for the entire data set, or you can group by one or more columns in the data set and compute an aggregate value for each group. The calculated aggregated column is defined in the data set being summarized, so every row in a group will have the same value in the calculated column (the aggregated value for that group). The column is hidden by default. You can choose to show the column or show its value in another control, which is what we do in the following section.

Example: Aggregating data with calculated fields

In this example, we will query the `SALES` table and create a `JdbTextField` component to display the sum of the `TOTAL_VALUE` field for the current `CUST_NO` field. To do this, we first create a new column called `GROUP_TOTAL`. Then set the `calcType` property of the column to `aggregated` and create an expression that summarizes the `TOTAL_VALUE` field from the `SALES` table by customer number and places the

resulting value in the GROUP_TOTAL column. The completed project is available in the /samples/DataExpress/Aggregating directory of your JBuilder installation.

- 1 Create a new application by following [“Retrieving data for the examples” on page 122](#).

This step enables you to connect to a database, read data from a table, and view and edit that data in a data-aware component.

- 2 Click `queryDataSet1` in the component tree.

This forms the query to populate the data set with values to be aggregated.

- 3 Open the `query` property of `queryDataSet1`, and modify the SQL Statement to read:

```
SELECT CUST_NO, PO_NUMBER, SHIP_DATE, TOTAL_VALUE from SALES
```

- 4 Click the Test Query button to test the query and ensure its validity, and when successful, click OK.

- 5 Click on the expand icon beside `queryDataSet1` in the component tree, select `<new column>`, and in the Inspector, set the following properties:

Property name	Value
<code>caption</code>	GROUP_TOTAL
<code>columnName</code>	GROUP_TOTAL
<code>currency</code>	True
<code>dataType</code>	BIGDECIMAL
<code>calcType</code>	aggregated
<code>visible</code>	Yes

A new column is instantiated and the following code is add to the `jbInit()` method. To view the code, select the Source tab. Select the Design tab to continue.

```
column1.setCurrency(true);
column1.setCalcType(com.borland.dx.dataset.CalcType.AGGREGATE);
column1.setCaption("GROUP_TOTAL");
column1.setColumnName("GROUP_TOTAL");
column1.setDataTypes(com.borland.dx.dataset.Variant.BIGDECIMAL);
```

- 6 Add a `JdbTextField` from the `dbSwing` page of the component palette to the UI designer, set its `dataSet` property to `queryDataSet1`, and set its `columnName` property to `GROUP_TOTAL`.

This control displays the aggregated data. You may wish to add a `JdbTextArea` to describe what the text field is displaying.

No data will be displayed in the `JdbTextField` in the designer. The calculations are only visible when the application is running. The data type of `BIGDECIMAL` is used here because that is the data type of the `TOTAL_VALUE` column which will be used in the calculation expression. Aggregated columns are always read-only.

- 7 Select each of the following columns, and set the visible property of each to `yes`.

- `PO_NUMBER`
- `CUST_NO`
- `SHIP_DATE`

This step ensures the columns that will display in the table are persistent. Persistent columns are enclosed in brackets in the content pane.

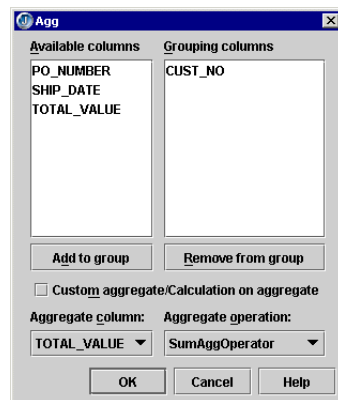
- 8 Select the GROUP_TOTAL column in the content pane, and to define the aggregation for this column, double-click the `agg` property to display the `agg` property editor.

In the `agg` property editor,

- a Select CUST_NO in the Available Columns list. Click Add to Group to select this as the field that will be used to define the group.
- b Select TOTAL_VALUE from the Aggregate Column list to select this as the column that contains the data to be aggregated.
- c Select `SumAggOperator` from the Aggregate Operation list to select this as the operation to be performed.

Based on above selections, you will have a sum of all sales to a given customer.

- 9 Click OK when the `agg` property editor looks like this:

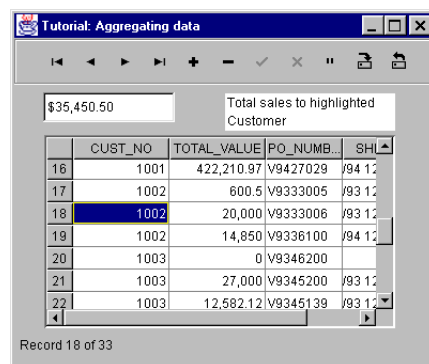


This step generates the following source code in the `jbInit()` method:

```
column1.setAgg(new com.borland.dx.dataset.AggDescriptor(new
    String[] { "CUST_NO" }, "TOTAL_VALUE", new
    com.borland.dx.dataset.SumAggOperator()));
```

- 10 Run the application by choosing Run/Run Project to view the aggregation results.

The running application looks like this:



When the application is running, the values in the aggregated field will automatically adjust to changes in the TOTAL_VALUE field. Also, the value that displays in the `JdbTextField` will display the aggregation for the CUST_NO of the currently selected row.

Setting properties in the AggDescriptor

The `agg` property editor provides a simple interface for creating and modifying `AggDescriptor` objects. An `AggDescriptor` object's constructor requires the following information:

- **Grouping Columns**—an array of strings (in any order) indicating the names of columns used to define a subset of rows of the `DataSet` over which the aggregation should occur.
- **Aggregate Column**—a string representing the name of the column whose values are to be aggregated.
- **Aggregate Operator**—name of an object of `AggOperator` type which performs the actual aggregate operation.

The `agg` property editor extracts possible column names for use as grouping columns, and presents them as a list of Available Columns. Only non-calculated, non-aggregate column names are allowed in the list of grouping columns.

If the `DataSet` for whose `Column` the `agg` property is being defined has a `MasterLink` descriptor (that is, if the `DataSet` is a detail `DataSet`), the linking column names will be added by default to the list of grouping columns when defining a new `AggDescriptor`.

The buttons beneath the list of grouping columns and available columns can be used to move the highlighted column name of the list above the button to the opposite list. Also, double-clicking on a column name in a list will move the column name to the opposite list. Entries within both lists are read-only. Since the ordering of column names is insignificant within a group, a column name is always appended to the end of its destination list. An empty (null) group is allowed.

The Aggregate Column choice control will contain the list of all non-aggregate column names for the current `DataSet`. Although the current set of `AggOperator` components provided with the `DataExpress` package does not provide support for non-numeric aggregate column types, we do not restrict columns in the list to numeric types, since it's possible that a user's customized `AggOperator` could support string and date types.

The Aggregate Operation choice control displays the list of `AggOperator` components built into `DataExpress` package as well as any user-defined `AggOperator` components within the same class context as the `AggDescriptor` component's `Column`.

Users desiring to perform calculations on aggregated values (for example, the sum of line items ordered multiplied by a constant) should check the Calculated Aggregate check box. Doing so disables the Aggregate Column and Aggregate Operation choice controls, and substitutes their values with "null" in the `AggDescriptor` constructor, signifying a calculated aggregate type. When the Calculated Aggregate check box is unchecked, the Aggregate Column and Aggregate Operation choice controls are enabled.

Creating a custom aggregation event handler

To use an aggregation method other than the ones provided by `JBuilder`, you can create a custom aggregation event handler. One way to create a custom aggregation event handler is to code the `calcAggAdd` and `calcAggDelete` events through the UI designer. `calcAggAdd` and `calcAggDelete` are `StorageDataSet` events that are called after the `AggOperator` is notified of an update operation.

A typical use for these events is for totaling columns in a line items table (like SALES). The dollar amounts can be totaled using a built-in `SumAggOperator`. Additional aggregated columns can be added with the `aggOperator` property for the `AggDescriptor` set to `null`. These additional columns might be for applying a tax or discount percentage on the subtotal, calculating shipping costs, and then calculating a final total.

You can also create a custom aggregation class by implementing a custom aggregation operator component by extending from `AggOperator` and implementing the abstract methods. The advantage of implementing a component is reusability in other `DataSet` components. You may wish to create aggregation classes for calculating an average, standard deviation, or variance.

Adding an Edit or Display Pattern for data formatting

All data stored internally, such as numbers and dates, is entered and displayed as text strings. *Formatting* is the conversion from the internal representation to a string equivalent. *Parsing* is the conversion from string representation to internal representation. Both conversions are defined by rules specified by string-based patterns.

All formatting and parsing of data in the `DataSet` package is controlled by the `VariantFormatter` class, which is uniquely defined for every `Column` in a `DataSet`. To simplify the use of this class, there are corresponding string properties which, when set, construct a `VariantFormatter` for the `Column` using the basic pattern syntax defined in the JDK `java.text.Format` classes.

There are four distinct kinds of patterns based on the data type of the item you are controlling.

- 1 Numeric patterns
- 2 Date and time patterns
- 3 String patterns
- 4 Boolean patterns

See “String-based patterns (masks)” in the *DataExpress Component Library Reference* for more information on patterns.

The `Column` level properties that work with these string-based patterns are:

- The `displayMask` property, which defines the pattern used for basic data formatting and data entry.
- The `editMask` property, which defines the pattern used for more advanced keystroke-by-keystroke data entry (also called parsing).
- The `exportDisplayMask` property, which defines the pattern used when importing and exporting data to text files.

The default `VariantFormatter` implementations for each `Column` are simple implementations which were written to be fast. Those columns using punctuation characters, such as dates, use a default pattern derived from the column’s locale. To override the default formatting (for example, commas for separating groups of thousands, or a decimal point), explicitly set the string-based pattern for the property you want to set (`displayMask`, `editMask`, or `exportDisplayMask`).

Setting a `displayMask`, `editMask`, or `exportDisplayMask` to an empty string or `null` has special meaning; it selects its pattern from the default `Locale`. This is the default behavior of JBuilder for columns of type `Date`, `Time`, `Timestamp`, `Float`, `Double`, and `BigDecimal`. By doing this, JBuilder assures that an application using the defaults will automatically select the proper display format when running under a different locale.

Note When writing international applications that use locales other than `en_US` (U.S. English locale), you must use the U.S. style separators (for example, the comma for the thousands separator and the period as the decimal point) in your patterns. This allows you to write an application that uses the same set of patterns regardless of its target locale. When using a locale other than `en_US`, these characters are translated by the JDK to their localized equivalents and displayed appropriately. For an example of using patterns in an international application, see the `IntlDemo.jpx` file, which is in the `/samples/dbSwing/MultiLingual` directory of your JBuilder installation.

To override the default formats for numeric and date values that are stored in locale files, set the `displayMask`, `editMask`, or `exportDisplayMask` property (as appropriate) on the `Column` component of the `DataSet`.

The formatting capabilities provided by `DataExpress` package string-based patterns are typically sufficient for most formatting needs. If you have more specific formatting needs, the format mechanism includes general-purpose interfaces and classes that you can extend to create custom format classes.

Display masks

Display masks are string-based patterns that are used to format the data displayed in the `Column`, for example, in a `JdbTable`. Display masks can add spaces or special characters within a data item for display purposes.

Display masks are also used to parse user input by converting the string input back into the correct data type for the `Column`. If you enter data which cannot be parsed using the specified display mask pattern, you will not be able to leave the field until data entered is correct.

Tip User input that cannot be parsed with the specified pattern generates validation messages. These messages appear on the `JdbStatusLabel` control when the `JdbStatusLabel` and the UI component that displays the data for editing (for example, a `JdbTable`) are set to the same `DataSet`.

Edit masks

Before editing starts, the display mask handles all formatting and parsing. Edit masks are optional string-based patterns that are used to control data editing in the `Column` and to parse the string data into the `Column` keystroke-by-keystroke.

In a `Column` with a specified edit mask, literals included in the pattern display may be optionally saved with the data. Positions in the pattern where characters are to be entered display as underscores (`_`) by default. As you type data into the `Column` with an edit mask, input is validated with each key pressed against characters that the pattern allows at that position in the mask.

Characters that are not allowed at a given location in the pattern are not accepted and the cursor moves to the next position only when the criteria for that location in the pattern is satisfied.

Using masks for importing and exporting data

When you import data into a `DataExpress` component, `JBuilder` looks for a `SCHEMA` file (`.schema`) by the same name as the data file. If it finds one, the settings in the `SCHEMA` file take precedence. If it doesn't find one, it looks at the column's `exportDisplayMask` property. Use the `exportDisplayMask` to format the data being imported.

Often, data files contain currency formatting characters which cannot be read directly into a numeric column. You can use an `exportDisplayMask` pattern to read in the values without the currency formatting. Once in `JBuilder`, set display and/or edit masks to re-establish currency (or other formatting) as desired.

When exporting data, `JBuilder` uses the `exportDisplayMask` to format the data for export. At the same time, it creates a `SCHEMA` file with these settings so that data can be easily imported back into a `DataExpress` component.

Data type dependent patterns

The following sections describe and provide examples for string-based patterns for various types of data.

Patterns for numeric data

Patterns for numeric type data consist of two parts: the first part specifies the pattern for positive numbers (numbers greater than 0) and the second for negative numbers. The two parts are separated with a semi-colon (;). The pattern symbols for numeric data are described in “Numeric data patterns” in the *DataExpress Component Library Reference*.

Numeric `Column` components always have display and edit masks. If you do not set these properties explicitly, default patterns are obtained using the following search order:

- 1 From the `Column` component’s locale.
- 2 If no locale is set for the `Column`, from the `DataSet` object’s locale.
- 3 If no locale is set for the `DataSet`, from the default system locale. Numeric data displays with three decimal places by default.

Numeric columns allow any number of digits to the left of the decimal point; however, masks restrict this to the number of digits specified to the left of the decimal point in the mask. To ensure that all valid values can be entered into a `Column`, specify sufficient digits to the left of the decimal point in your pattern specification.

In addition, every numeric mask has an extra character positioned at the left of the data item that holds the sign for the number.

The code that sets the display mask to the first pattern in the table below is:

```
column1.setDisplayMask(new String("###%"));
```

The following table explains several sample pattern specifications for numeric data:

Pattern specification	Data values	Formatted value	Meaning
###%	85	85%	All digits are optional, leading zeros do not display, value is divided by 100 and shown as a percentage
#,##0.0#^ cc;-#,##0.0#^ cc	500.0 -500.5 004453.3211 -00453.3245	500.0 cc -500.5 cc 4,453.32 cc -453.32 cc	The “0” indicates a required digit, zeroes are not suppressed. Negative numbers are preceded with a minus (–) sign. The literal “cc” displays beside the value. The cursor is positioned at the point of the carat (^) with digits moving to the left as you type each digit.
\$#,###.##;(\$#,###.##)	4321.1 -123.456	\$4,321.1 (\$123.46)	All digits optional, includes a thousands separator, decimal separator, and currency symbol. Negative values enclosed in parenthesis. Typing in a minus sign (–) or left parenthesis (()) causes JBuilder to supply parenthesis surrounding the value.

Patterns for date and time data

Columns that contain date, time, and timestamp data always have display and edit masks. If you do not set these properties explicitly, default patterns are obtained using the following search order:

- 1 From the `Column` component’s locale.
- 2 If no locale is set for the `Column`, from the `DataSet` object’s locale.
- 3 If no locale is set for the `DataSet`, from the default system locale.

The pattern symbols you use for date, time, and timestamp data are described in “Date/time data patterns” in the *DataExpress Component Library Reference*.

For example, the code that sets the edit mask to the first pattern listed below is:

```
column1.setDisplayMask(new String("MMM dd, yyyyG"));
```

The following table explains several sample pattern specifications for date and time data:

Pattern specification	Data values	Formatted value	Meaning
MMM dd, yyyyG	January 14, 1900 February 2, 1492	Jan 14, 1900AD Feb 02, 1492AD	Returns the abbreviation of the month, space (literal), two digits for the day, 4 digits for year, plus era designator
MM/d/yy H:m	July 4, 1776 3:30am March 2, 1997 11:59pm	07/4/76 3:30 03/2/92 23:59	Returns the number of the month, one or two digits for the day (as applicable), two digits for the year, plus the hour and minute using a 24-hour clock

Patterns for string data

Patterns for formatting and editing text data are specific to DataExpress classes. They consist of up to four parts, separated by semicolons, of which only the first is required. These parts are:

- 1 The string pattern.
- 2 Whether literals should be stored with the data or not. A value of 1 indicates the default behavior, to store literals with the data. To remove literals from the stored data, specify 0.
- 3 The character to use as a blank indicator. This character indicates the spaces to be entered in the data. If this part is omitted, the underscore character is used.
- 4 The character to use to replace blank positions on output. If this part is omitted, blank positions are stripped.

The pattern symbols you use for text data are described in “Text data patterns” in the *DataExpress Component Library Reference*.

For example, the code that sets the display and edit masks to the first pattern listed below is:

```
column1.setDisplayMask(new String("00000{-9999}"));
column1.setEditMask(new String("00000{-9999}"));
```

The following table explains some pattern specifications:

Pattern specification	Data values	Formatted value	Meaning
00000{-9999}	950677394 00043 1540001	95067-7394 00043 00154-0001	Display leading zeros for the left 5 digits (required), optional remaining characters include a dash literal and 4 digits. Use this pattern for U.S. postal codes.
L0L 0L0	H2A2R9 M1M3W4	H2A 2R9 M1M 3W4	The L specifies any letter A-Z, entry required. The 0 (zero) specifies any digit 0-9, entry required, plus (+) and minus (-) signs not permitted. Use this pattern for Canadian postal codes.
{{999}} 000-0000^!;0	4084311000	(408) 431-1000	A pattern for a phone number with optional area code enclosed in parenthesis. The carat (^) positions the cursor at the right side of the field and data shifts to the left as it is entered. To ensure data is stored correctly from right to left, use the ! symbol. (Numeric values do this automatically.) The zero (0) indicates that literals are not stored with the data.

Patterns for boolean data

The `BooleanFormat` component uses a string-based pattern that is helpful when working with values that can have two values, stored as `true` or `false`. Data that falls into each category is formatted using string values you specify. This formatter also has the capability to format `null` or unassigned values.

For example, you can store gender information in a column of type `boolean` but can have JBuilder format the field to display and accept input values of “Male” and “Female” as shown in the following code:

```
column1.setEditMask("Male;Female;");
column1.displayMask("Male;Female;");
```

The following table illustrates valid boolean patterns and their formatting effects:

Pattern specification	Format for true values	Format for false values	Format for null values
male;female	male	female	(empty string)
T,F,T	T	F	T
Yes,No,Don't know	Yes	No	Don't know
smoker;;	smoker	(empty string)	(empty string)
smoker;nonsmoker;	smoker	nonsmoker	(empty string)

Presenting an alternate view of the data

You can sort and filter the data in any `StorageDataSet`. However, there are situations where you need the data in the `StorageDataSet` presented using more than one sort order or filter condition simultaneously. The `DataSetView` component provides this capability.

The `DataSetView` component also allows for an additional level of indirection which provides for greater flexibility when changing the binding of your UI components. If you anticipate the need to rebind your UI components and have several of them, bind the components to a `DataSetView` instead of directly to the `StorageDataSet`. When you need to rebind, change the `DataSetView` component to the appropriate `StorageDataSet`, thereby making a single change that affects all UI components connected to the `DataSetView` as well.

To create a `DataSetView` object, and set its `storageDataSet` property to the `StorageDataSet` object that contains the data you want to view,

- 1 Create a new application by following [“Retrieving data for the examples” on page 122](#).
This step enables you to connect to a database, read data from a table, and view and edit that data in a data-aware component.
- 2 Add a `DataSetView` component from the DataExpress page to the component tree or the UI designer.
- 3 Set the `storageDataSet` property of the `DataSetView` component to `queryDataSet1`.
The `DataSetView` navigates independently of its associated `StorageDataSet`.
- 4 Add another `TableScrollPane` and `JdbTable` to the UI designer, and to enable the controls to navigate together, set the `dataSet` property of the `JdbTable` to `dataSetView1`.
- 5 Compile and run the application.

The `DataSetView` displays the data in the `QueryDataSet` but does not duplicate its storage. It presents the original unfiltered and unsorted data in the `QueryDataSet`.

You can set filter and sort criteria on the `DataSetView` component that differ from those on the original `StorageDataSet`. Attaching a `DataSetView` to a `StorageDataSet` and setting new filter and/or sort criteria has no effect on the filter or sort criteria of the `StorageDataSet`.

To set filter and/or sort criteria on a `DataSetView`,

- 1 Double-click the Frame file in the project pane, and select the Design tab.
- 2 Select the `DataSetView` component.
- 3 On the Properties page in the Inspector,
 - a Select the `sort` property to change the order records are displayed in the `DataSetView`.
See [“Sorting data” on page 127](#) for more information on the `sortDescriptor`.
 - b Select the `masterLink` property to define a parent data set for this view.
See [Chapter 9, “Establishing a master-detail relationship”](#) for more information on the `masterLinkDescriptor`.
- 4 On the Events page in the Inspector, select the `filterRow` method to temporarily hide rows in the `DataSetView`. See [“Filtering data” on page 124](#) for more information on filtering.

You can edit, delete, and insert data in the `DataSetView` by default. When you edit, delete, and insert data in the `DataSetView`, you are also editing, deleting, and inserting data in the `StorageDataSet` the `DataSetView` is bound to.

- Set the `enableDelete` property to `false` to disable the user’s ability to delete data from the `StorageDataSet`.
- Set the `enableInsert` property to `false` to disable the user’s ability to insert data into the `StorageDataSet`.
- Set the `enableUpdate` property to `false` to disable the user’s ability to update data in the `StorageDataSet`.

Ensuring data persistence

Between the time that you develop an application and each time the user runs it, many changes can happen to the data at its source. Typically, the data within the data source is updated. But more importantly, structural changes can happen and these types of changes cause greater risk for your application to fail. When such condition occurs, you can

- Let the running application fail, if and when a such event is encountered. For example, a lookup table’s column gets renamed at the database server but this is not discovered until an attempt is made in the application to edit the lookup column.
- Stop the application from running and display an error message. Depending on where the unavailable data source is encountered, this approach reduces the possibility of partial updates being made to the data.

By default, the columns that display in a data-aware component are determined at run-time based on the `Columns` that appear in the `DataSet`. If the data structure at the data source has been updated and is incompatible with your application, a run-time error is generated when the situation is encountered.

JBuilder offers support for data persistence as an alternative handling of such situations. Use this feature if your application depends on particular columns of data being available for your application to run properly. This assures that the column will be there and the data displayed in the specified order. If the source column of the

persistent `Column` changes or is deleted, an `Exception` is generated instead of a run-time error when access to the column's data fails.

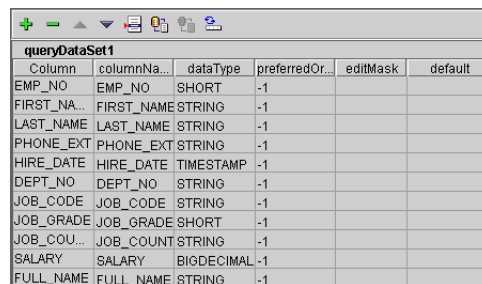
Making columns persistent

You can make a column persistent by setting any property at the `Column` level (for example, an edit mask). When a column has become persistent, square brackets ([]) are placed around the column name in the component tree.

To set a `Column` level property,

- 1 Open any project that includes a `DataSet` object, for example, select any project file (.jpx) in the `/samples/DataExpress/` directory of your JBuilder installation.
- 2 Double-click the Frame file (for example, `BasicAppFrame.java`) to open it into the content pane, then click the Design tab.
- 3 Double-click the `DataSet` object. This displays the column designer for the data set. The column designer looks like this for the employee sample table:

Figure 12.3 Column designer



Column	columnName	dataType	preferredOr...	editMask	default
EMP_NO	EMP_NO	SHORT	-1		
FIRST_NAME	FIRST_NAME	STRING	-1		
LAST_NAME	LAST_NAME	STRING	-1		
PHONE_EXT	PHONE_EXT	STRING	-1		
HIRE_DATE	HIRE_DATE	TIMESTAMP	-1		
DEPT_NO	DEPT_NO	STRING	-1		
JOB_CODE	JOB_CODE	STRING	-1		
JOB_GRADE	JOB_GRADE	SHORT	-1		
JOB_COUNT	JOB_COUNT	STRING	-1		
SALARY	SALARY	BIGDECIMAL	-1		
FULL_NAME	FULL_NAME	STRING	-1		

- 4 Select the `Column` for which you want to set the property. The Inspector updates to reflect the properties (and events) of the selected column.
- 5 Set any property by entering a value in its value box in the Inspector. If you don't want to change any column properties, you can set a value, then reset the value to its default.

To demonstrate, set the a minimum value for a `Column` containing numeric data by entering a numeric value in the `min` property. JBuilder automatically places square brackets ([]) around the column name in the component tree.

In the column designer, the columns for that data set are displayed in a table in the UI designer. A toolbar for adding, deleting, and navigating the data set is provided.

- The Insert Column into the DataSet button inserts a new column at the preferred ordinal of the highlighted column in the table.
- The Delete button removes the column from the data set.
- The Move Up and Move Down buttons manipulate the columns preferred ordinal, changes the order of display in data-aware components, such as a table control.
- The Choose The Properties To Display button lets you choose which properties to display in the designer.
- The Persist All MetaData button will persist all the metadata that is needed to open a `QueryDataSet` at run time. See [“Using the column designer to persist metadata” on page 71](#).

- The Make All MetaData Dynamic button enables you to update a query after the table may have changed on the server. You must first make the metadata dynamic, then persist it, to use new indices created on the database table. Selecting Make All MetaData Dynamic will REMOVE CODE from the source file. See [“Making metadata dynamic using the column designer” on page 72](#).
- The Generate RowIterator Class button opens a dialog that provides lightweight (low memory usage and fast binding) iteration capabilities to ensure static type-safe access to columns. See [“Generate RowIterator Class button” on page 71](#) for more information.

Note If you are using JDataStore for database access, you can use the JDataStore Explorer (Tools|JDataStore Explorer) to restructure the JDataStore database from within JBuilder and without writing any code.

To close the column designer, double-click any UI component, such as `contentPane`, in the component tree. The only way to close one designer is to open a different one.

Using variant data types

Columns can contain many types of data. This topic discusses storing Java objects in a Column. Columns are introduced more completely in [Chapter 7, “Working with columns.”](#)

Storing Java objects

`DataSet` and `DataStore` can store Java objects in columns of a `DataSet`. Fields in a SQL table, reported by JDBC as being of type `java.sql.Types.OTHER`, are mapped into columns whose data type is `Variant.OBJECT`, or you can set a column's data type to Object and set/get values through the normal data set API.

If a `DataStore` is used, the objects must be serializable. If they are not, an exception is raised whenever the `DataStore` attempts to save the object. Also, the class must exist on the CLASSPATH when it attempts to read an object. If not, the attempt will fail.

Note the following about formatting and editing a column that contains a Java object:

- Default formatting and editing

In the UI designer, a formatter is assigned to `Object` columns by default. When the object is edited, it will simply be an object of type `java.lang.String` regardless of what the type was originally.

- Custom formatting and editing

You can, and probably will want to, define the `formatter` property on a column to override the default functionality, or at least make the column non-editable. You can use a custom formatter to define the proper formatting and parsing of the objects kept in the column.

A column formatter is used for all the records in the data set. The implication of this is that you cannot mix object types in a particular column. This restriction is only for customized editing.

Chapter 13

Using other controls and events

This topic provides more information on using controls and events. [“Retrieving data for the examples” on page 122](#) sets up a query that can be used as a starting point for any of the discussions in this chapter.

Topics discussed in this chapter include,

- [“Synchronizing visual components” on page 157](#)
- [“Accessing data and model information from a UI component” on page 158](#)
- [“Displaying status information” on page 158](#)
- [“Handling errors and exceptions” on page 159](#)

Synchronizing visual components

Several data-aware components can be associated with the same `DataSet`. In such cases, the components navigate together. When you change the row position of a component, the row position changes for all components that share the same cursor. This synchronization of components that share a common `DataSet` can greatly ease the development of the user-interface portion of your application.

The `DataSet` manages a *pseudo* record, an area in memory where a newly inserted row or changes to the current row are temporarily stored. Components which share the same `DataSet` as their data source share the same pseudo record. This allows updates to be visible as soon as entry at the field level is complete, such as when you navigate off the field.

You synchronize multiple visual components by setting each of their `dataSet` properties to the same data set. When components are linked to the same data set, they *navigate* together and will automatically stay synchronized to the same row of data. This is called *shared cursors*.

For example, if you use a `JdbNavToolBar` and a `JdbTable` in your program, and connect both to the same `QueryDataSet`, clicking the “Last” button of the `JdbNavToolBar` automatically displays the last record of the `QueryDataSet` in the `JdbTable` as well. If those components are set to different `dataSet` components, they do not reposition automatically to the same row of data. Several of the `dbSwing` components, including `JdbNavToolBar` and `JdbStatusLabel`, automatically attach themselves to whichever `DataSet` has focus.

The `goToRow(com.borland.dx.dataset.ReadRow)` method provides a way of synchronizing two `DataSet` components to the same row (the one that `dataSet` is on) even if different sort or filter criteria are in effect.

Accessing data and model information from a UI component

If you set the `dataSet` property on a component, you should avoid accessing the `DataSet` data or model information programmatically through the component until the component's peer has been created; basically, this means until the component is displayed in the application UI.

Operations which fail or return incorrect/inconsistent results when executed before the component is displayed in the application UI include any operation that accesses the model of the component. This may include,

- `<component>.get()` or `<component>.set()` operations
- `<component>.insertRow()`
- and so on.

To assure successful execution of such operations, check for the `open()` event notification generated by the `DataSet`. Once the event notification occurs, you are assured that the component and its model are properly initialized.

Displaying status information

Many data applications provide status information about the data in addition to displaying the data itself. For example, a particular area of a window often contains information on the current row position, error messages, and other similar information. `dbSwing` includes a `JdbStatusLabel` component which provides a mechanism for such status information. It has a `text` property that allows you to assign a text string to be displayed in the `JdbStatusLabel`. This string overwrites the existing contents of the `JdbStatusLabel` and is overwritten itself when the next string is written to the `JdbStatusLabel`.

The `JdbStatusLabel` component automatically connects to whichever `DataSet` has focus. The `JdbStatusLabel` component doesn't display the data from the `DataSet`, but displays the following status information generated by the `DataSet`:

- Current row position
- Row count
- Validation errors
- Data update notifications
- Locate messages

Building an application with a JdbStatusLabel component

This section describes how to use the JBuilder design tools to add a `JdbStatusLabel` `dbSwing` component to an application.

To add the `JdbStatusLabel` to the UI of your existing application,

- 1 Open the project files for the application to which you want to add a `JdbStatusLabel`.

This application should include a `JdbTable` component, a `Database` component, and a `QueryDataSet` component. If you do not have such an application, use the files created for the section [“Retrieving data for the examples” on page 122](#). Make sure the layout for the project's `contentPane` is set to `null`.

- 2 Double-click the `Frame` file in the project pane of the IDE to open it in the content pane, then click the `Design` tab that appears at the bottom of the IDE.
- 3 Click the `dbSwing` page of the component palette and click the `JdbStatusLabel` component.

- 4 Draw the `JdbStatusLabel` below the `JdbTable` component. `jdbStatusLabel1` component automatically connects to whichever `DataSet` object has focus.

You typically use a `JdbStatusLabel` component in conjunction with another UI component, usually a `JdbTable` that displays the data from the `DataSet`. This sets both components to track the same `DataSet` and is often referred to as a shared cursor.

Once the `JdbStatusLabel` is added, you'll notice that the `JdbStatusLabel` component displays information that the cursor is on Row 1 of x (where x is the number of records in the `DataSet`).

- 5 Double-click the `QueryDataSet`.

This displays the column designer.

- 6 Select the `Last_Name` and `First_Name` columns and set the `required` property to `true` for both in the Inspector.
- 7 Set the `SALARY` column's `min` property to 25000.
- 8 Run the application.

Running the `JdbStatusLabel` application

When you run the application, you'll notice that when you navigate the data set, the row indicator updates to reflect the current row position. Similarly, as you add or delete rows of data, the row count is updated simultaneously as well.

To test its display of validation information,

- 1 Insert a new row of data. Attempt to post this row without having entered a value for the `FIRST_NAME` or `LAST_NAME` columns. A message displays in the `JdbStatusLabel` indicating that the row cannot be posted due to invalid or missing field values.
- 2 Enter a value for the `FIRST_NAME` and `LAST_NAME` columns. Enter a number in the `SALARY` column (1000) that doesn't meet the minimum value. When you attempt to move off the row, the `JdbStatusLabel` displays the same message that the row cannot be posted due to invalid or missing field values.

By setting the text of the `JdbStatusLabel` at relevant points in your program programmatically, you can overwrite the current message displayed in the `JdbStatusLabel` with your specified text. This text message, in turn, gets overwritten when the next text is set or when the next `DataSet` status message is generated. The status message can result from a navigation through the data in the table, validation errors when editing data, and so on.

Handling errors and exceptions

With programmatic usage of the `DataExpress` classes, most error handling is surfaced through `DataExpress` extensions of the `java.lang.Exception` class. All `DataSet` exception classes are of type `DataSetException` or its subclass.

The `DataSetException` class can have other types of exceptions chained to them, for example, `java.io.IOException` and `java.sql.SQLException`. In these cases, the `DataSetException` has an appropriate message that describes the error from the perspective of a higher level API. The `DataSetException` method `getExceptionChain()`

can be used to obtain any chained exceptions. The chained exceptions (a singly linked list) are non-`DataSetException` exceptions that were encountered at a lower-level API.

The `dataset` package has some built-in `DataSetException` handling support for dbSwing data-aware components. The controls themselves don't know what a `DataSetException` is. They simply expect all of their data update and access operations to work, leaving the handling of errors to the built-in `DataSetException`.

For dbSwing data-aware components, the default `DataSetException` error handling works as follows:

- If a control performs an operation that causes a `DataSetException` to occur, an Exception dialog box is presented with the message of the error. This Exception dialog box has a Details button that displays the stack trace.
- If the `DataSetException` has chained exceptions, they can be viewed in the Exception dialog box using the Previous and Next buttons.
- If the exception thrown is `ValidationException` (a subclass of `DataSetException`), the Exception dialog box displays only if there are no `StatusEvent` listeners on the `DataSet`, for example, the `JdbStatusLabel` control. A `ValidationException` is generated by a constraint violation, for example, a minimum or maximum value outside specified ranges, a data entry that doesn't meet an edit mask specification, an attempt at updating a read-only column, and so on. If a `JdbStatusLabel` control is bound to a `DataSet`, it automatically becomes a `StatusEvent` listener. This allows users to see the messages resulting from constraint violations on the status label.

Overriding default `DataSetException` handling on controls

You can override part of the default error handling by registering a `StatusEvent` listener with the `DataSet`. This prevents `ValidationException` messages from displaying in the Exceptions dialog box.

The default `DataSetException` handling for controls can be further disabled at the `DataSet` level by setting its `displayErrors` property to `false`. Because this is a property at the `DataSet` level, you need to set it for each `DataSet` in your application to effectively disable the default error handling for all `DataSet` objects in your application.

To completely control `DataSetException` handling for all dbSwing controls and `DataSet` objects, create your own handler class and connect it to the `ExceptionEvent` listener of the `DataSetException` class.

Most of the events in the `dataset` package throw a `DataSetException`. This is very useful when your event handlers use `dataset` APIs (which usually throw `DataSetException`). This releases you from coding `try/catch` logic for each event handler you write. Currently the JBuilder design tools do not insert the “throws `DataSetException`” clause in the source java code it generates, however you can add the clause yourself.

Chapter 14

Creating a distributed database application using DataSetData

The `DataSetData.jpx` sample project in the `/samples/DataExpress/StreamableDataSets` directory of your JBuilder installation contains a completed distributed database application using Java Remote Method Invocation (RMI) and `DataSetData`. It includes a server application that will take data from the sample `JDataStore` employee table and send the data via RMI in the form of `DataSetData`. A `DataSetData` is used to pass data as an argument to an RMI method or as an input stream to a Java servlet.

A client application will communicate with the server through a custom *Provider* and a custom *Resolver*. The client application displays the data in a table. Editing performed on the client can be saved using a `JdbNavToolBar` **Save** button.

For more information on writing custom providers, see [“Writing a custom data provider” on page 65](#). For information on writing or customizing a resolver, see [“Customizing the default resolver logic” on page 87](#).

See the file `DataSetData.html` in the `/samples/DataExpress/StreamableDataSets/` directory for updated information on this sample application.

Understanding the sample distributed database application (using Java RMI and DataSetData)

The sample project, found in `/samples/DataExpress/StreamableDataSets/DataSetData.jpx`, demonstrates the use of the `DataExpress DataSetData` class to build a distributed database application. In addition to using `DataSetData` objects to pass database data between an RMI server and client, this sample illustrates the use of a custom `DataSet Provider` and `Resolver`. The sample application contains the following files:

- Interface files

`EmployeeApi.java` is an interface that defines the methods we want to remote.

- Server files

`DataServerApp.java` is an RMI server. It extends `UnicastRemoteObject`.

- **Provider files**

`ClientProvider.java` is an implementation of a **Provider**. The `provideData` method is an implementation of a method in `com.borland.dx.dataset.Provider`. We look up the “DataServerApp” service on the host specified by the `hostName` property, then make the remote method call and load our `DataSet` with the contents.

- **Resolver files**

`ClientResolver.java` is an implementation of a **Resolver**. The `resolveData` method is an implementation of `com.borland.dx.dataset.Resolver`. First, we look up the “DataServerApp” service on the host specified by the `hostName` property. Then, we extract the changes into a `DataSetData` instance. Next, we make the remote method call, handle any resolution errors, and change the status bits for all changed rows to show that they have been resolved.

- **Client files**

`ClientApp.java` is an RMI client application. See `ClientFrame.java` for details.

- **Other files**

`Res.java` is a resource file for internationalizing the application.

`ClientFrame.java` is the frame of `ClientApp`. Notice that the `DataSet` displayed in the table is a `TableDataSet` with a custom provider and a custom resolver. See `ClientProvider.java` and `ClientResolver.java` for details.

`DataServerFrame.java` is the frame displayed by `DataServerApp`.

Setting up the sample application

To run the sample application, you need to perform the following steps:

- 1 Open this application in JBuilder by choosing **File|Open** and browsing to `/samples/DataExpress/StreamableDataSets/DataSetData.jpx`.
- 2 Choose **Project|Project Properties**, and set the runtime options for the application:
 - a Select the **Run** tab.
 - b Select the **Run DataServerApp** runtime configuration (the default), and click **Edit** to open the **Runtime Configuration Properties** dialog box.
 - c Check that the properties in the **VM Parameters** field have the correct path.
 The `java.rmi.server.codebase` property points to the location of the RMI server’s classes. By default, this property is set as follows:


```
-Djava.rmi.server.codebase="file:
    /usr/local/jbuilder/samples/DataExpress/StreamableDataSets/classes/
    file:/usr/local/jbuilder/lib/dx.jar"
```

 The `java.rmi.server.codebase` property points to the proper location of the RMI server’s classes. By default, this property is set as follows:


```
-Djava.security.policy=file:/usr/local/jbuilder/samples/DataExpress/
    StreamableDataSets/SampleRMI.policy
```
 - d Close the **Project Properties** dialog box.
- 3 Start the RMI registry by choosing **Tools|RMIRegistry** from JBuilder.
 The registry is toggled on and off from the **Tools** menu.
- 4 Right-click `DataServerApp.java` in the project pane, and choose **Run|Run DataServerApp** to start the RMI server.

- 5 Right-click `ClientApp.java` in the project pane, and choose `Run!Run ClientApp` to start the RMI client.
- 6 Edit the data in the `ClientApp`'s table, and click either the `Save Changes` button on the toolbar (to save the changes back to the server) or the `Refresh` button on the toolbar (to reload the data from the server).

Each time data is saved or refreshed, the middle-tier request counter increases.

What is going on?

`DataServerApp` registers itself as an RMI server. It responds to two RMI client requests: `provideEmployeeData` and `resolveEmployeeChanges`, as defined in the RMI remote interface `EmployeeApi.java`.

The client application consists of a frame (`ClientFrame.java`) with `JdbTable` and `JdbNavToolBar` `dbSwing` components for displaying data in a `DataExpress TableDataSet`. Data is provided to the `TableDataSet` via a *custom provider*, `ClientProvider.java`, and data is saved to the source via a *custom resolver*, `ClientResolver.java`. `ClientProvider.java` fills its table data by invoking the `DataServerApp provideEmployeeData()` remote method via RMI. This causes `DataServerApp` to query data from a table on a JDBC database server into a `DataSet`. It then extracts the data from the `DataSet` into a `DataSetData` object and sends it back to `ClientProvider` via RMI. `ClientProvider` then loads the data in the `DataSetData` object into the `ClientApp DataSet`, and the data appears in the table.

When it is time to resolve changes made in the table back to the database, the `ClientApp DataSet custom resolver`, `ClientResolver.java`, extracts (only) the changes that need to be sent to the database server into a `DataSetData` object. `ClientResolver` then invokes the `DataServerApp resolveEmployeeChanges()` remote method via RMI, passing it the `DataSetData` object containing the necessary updates as the parameter.

`DataServerApp` then uses `DataExpress` to resolves the changes back to the database server. If an error occurs (due to a business rule or data constraint violation, for example) `DataServerApp` packages rows which could not be saved back to the database into a `DataSetData` object and returns it back to `ClientResolver`. `ClientResolver` then extracts the unresolvable rows in the `DataSetData` object into the `ClientApp` table, allowing the problematic rows to be corrected and resolved back to the server again.

Note that `DataServerApp` is the *middle-tier* of the application. It can enforce its own business rules and constraints between the database server and the client. And, of course, it could provide any number of additional remotely accessibly methods for implementing business logic or application-specific tasks.

Passing metadata by DataSetData

The metadata passed in a `DataSetData` object is very limited. Only the following `Column` properties are passed:

- `columnName`
- `dataType`
- `precision`
- `scale`
- `hidden`
- `rowId`

Other column properties that a server needs to pass to a client application, should be passed as an array of `Columns` via RMI. The `Column` object itself is serializable, so a client application could be designed to get these column properties before it needed the data. The columns should be added as persistent columns before the `DataSetData` is loaded.

Deploying the application on multiple tiers

To deploy the application on multiple tiers,

- 1 Open `DataServerApp.java`, and modify the database connection URL in the constructor to point to a remote database connection to which you have access.

The database is the back end, or third tier.

- 2 Choose **Project!Make Project** to recompile and update the `DataServerApp` class.
- 3 Deploy `DataServerApp.class` to a remote machine to which you are connected.
`DataServerApp` runs on the middle, or second, tier.
- 4 Start the RMI Registry on the middle tier computer.
- 5 Start `DataServerApp` on the middle tier.

Note Beginning with JDK 1.2, it is necessary to grant an RMI server special security rights in order for it to listen for and accept client RMI requests over a network. Typically, these rights are specified in a Java security policy file defined by a special property, `java.security.policy`, passed by way of a command-line argument to the VM of the server. This is similar to the `java.rmi.server.codebase` property which must also be passed to the server's VM. A sample RMI security policy file which will allow an RMI client to connect to the server is included with this project in the file `SampleRMI.policy`.

When starting `DataServerApp` on the middle-tier, make sure both the `java.security.policy` and `java.rmi.server.codebase` properties are set to the proper locations on the middle-tier machine.

- 6 Double-click `ClientFrame.java` in the project pane of JBuilder to open it in the content pane. Select the Design tab to invoke the designer. Select `clientProvider1` in the component tree and modify the `hostName` property to the hostname of the middle-tier machine.
- 7 Select `clientResolver1` and modify the `hostName` property to the hostname of the middle-tier machine.
- 8 Choose **Project!Make Project** to rebuild `ClientApp`.

Start `ClientApp` on the client, or first tier, by right-clicking on the `ClientApp.java` file in the project pane and choosing Run.

For more information

- Read the RMI Documentation on the Sun web site at <http://java.sun.com/j2se/1.4/docs/guide/rmi/>.
- Learn more about writing custom *Providers* and *Resolvers* by viewing the sample data set application `/samples/DataExpress/CustomProviderResolver/CustomProviderResolver.jpx`.

Chapter 15

Database administration tasks

**This is a feature of
JBuilder Developer
and Enterprise**

This chapter provides information on how to accomplish some common database administrator tasks. The following subjects are covered:

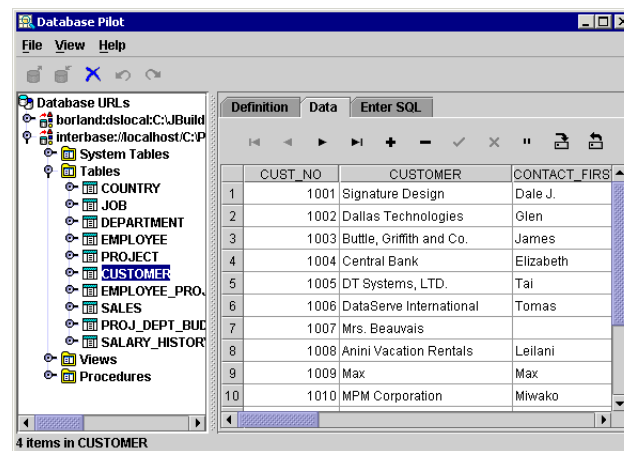
- “Exploring database tables and metadata using the Database Pilot” on page 165
- “Using the Database Pilot for database administration tasks” on page 170
- “Monitoring database connections” on page 172

Exploring database tables and metadata using the Database Pilot

The Database Pilot is a hierarchical database browser that also allows you to view and edit data. It presents JDBC-based metadata for databases in a two-pane window. The left pane contains a tree that hierarchically displays a set of databases and their associated tables, views, stored procedures, and metadata. The right pane is a multi-page display of descriptive information for each node of the tree. In certain cases, you can edit data in the right pane as well.

To display the Database Pilot, choose Tools|Database Pilot from the JBuilder menu.

Figure 15.1 Database Pilot



Through a persistent connection to a database, the Database Pilot enables you to:

- Browse database schema objects, including tables, table data, columns (fields), indexes, primary keys, foreign keys, stored procedure definitions, and stored procedure parameters.
- View, create, and modify database URLs.
- Enter and execute SQL statements to query a database.
- Create, view, and edit data in existing tables.

Browsing database schema objects

The Database Pilot window contains a menu, a toolbar, a status label, and two panes of database information.

- The left pane displays a hierarchical tree of objects that include database URLs, tables (and their columns, indexes, primary key, and foreign keys), views, system tables, and stored procedures (and their parameters).

An expand icon beside an object in the left pane indicates that the object contains other objects below it. To see those objects, click the expand icon. When an object is expanded to show its child objects, the expand icon becomes a contract icon. To hide child objects, click the contract icon.

- The right pane contains tabbed pages that display the contents of objects highlighted in the left pane. The tabbed pages in the right pane vary depending on the type of object highlighted in the left pane. For example, when a database alias is highlighted in the left pane, the right pane displays a Definition page that contains the database URL, Driver, UserName, and other parameters, or properties. Bold parameter names indicate a parameter that cannot be modified. All other parameters that appear in the right pane can be edited there. The following tabbed pages may appear in the right hand pane:

- Definition
- Enter SQL
- Summary
- Data

For more information, launch the Database Pilot by choosing Tools|Database Pilot from the menu, and refer to its online help.

Setting up drivers to access remote and local databases

The Database Pilot browses databases listed in the Connection URL History List section of the `<home>/dbpilot/dbpilot.properties` file. Additions are made to this list when you connect to a database using the `connection` property editor of a Database component.

You can use the Database Pilot to view, create, modify, and delete database URLs. The following procedures assume the URL is closed, and lists each task, briefly describing the steps needed to accomplish it.

View URL

To view a URL,

- 1 In the left pane, select the URL to view.
The Definition page appears in the right pane.
- 2 Click the expand icon beside a database URL (or double-click it) in the left pane to see its contents.

Create URL

To create a URL,

- 1 Right-click a URL or database in the left pane to open the context menu.
- 2 Choose New from the context menu (or choose New from the File menu) to open the New URL dialog box.
- 3 Select a Driver from the drop-down list or enter the driver information.
Drivers must be installed to be used, and the driver's files must be listed in the CLASSPATH statement in the JBuilder setup script. This is done on the Database Drivers page of the Enterprise Setup dialog box (Tools|Enterprise Setup).
- 4 Browse to or enter the desired URL and click OK.
- 5 On the Definition page in the right pane, specify the UserName and any other desired properties.
- 6 Click the Apply button on the toolbar to apply the connection parameters.

Modify URL

To modify an existing URL,

- 1 Select the URL to modify in the left pane.
- 2 Edit settings on the Definition page as desired.
- 3 Click the Apply button on the toolbar to update the connection parameters, or click the Cancel button to undo changes to the settings.

Delete URL

To delete a URL,

- 1 Select the URL to delete in the left pane.
- 2 Choose File|Delete to remove the URL.

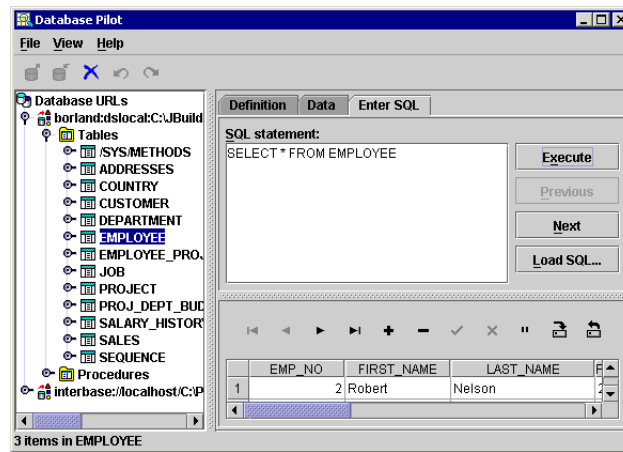
Note If you're creating a new ODBC URL and you are running Windows NT, you must define its ODBC Data Source through the Windows Control Panel before you can connect to that database.

Executing SQL statements

The Enter SQL page displays a window in which you can enter SQL statements, or specify and execute an existing .SQL file. The main part of the screen is an edit box where you can enter SQL statements. To the right of the edit box are four buttons, the Execute button, the Next button, the Previous button, and the Load SQL button. When a SQL SELECT statement is executed, the results of the query are displayed in an

editable table, which is located below the edit box. This screen may need to be resized to view all its components. The page looks like this:

Figure 15.2 Enter SQL page of the Database Pilot



To query a database using SQL,

- 1 Open a database by selecting its URL in the left pane and entering a user name and password, if applicable.
- 2 Select the database or one of its child nodes in the left pane.
- 3 Click the Enter SQL tab in the right pane to display an edit box where you can enter or select a SQL statement.
- 4 Enter (or paste) a SQL statement in the edit box, or click the Load SQL button and enter a SQL file name.

If you enter non-SELECT statements, the statement is executed, but no result set is returned.

- 5 Click the Execute button to execute the query.

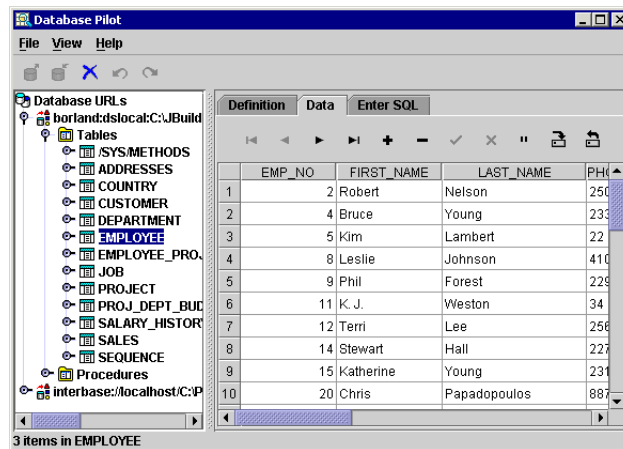
You can copy SQL statements from text files, a Help window, or other applications and paste them into the edit box. Some SQL servers require that the table name be entered in quotation marks, some do not require this.

Note If the SQL syntax you enter is incorrect, an error message is generated. You can freely edit the Enter SQL field to correct syntax errors.

Using the Explorer to view and edit table data

Select the Data page to display the data in a selected table, view, or synonym. You can enter and edit records in a table on the Data page if the table permits write access, and if the Request Updatable Queries box of the Query page of the ViewOptions menu is checked. The Data page displays a table populated with the data from the selected

table. A toolbar control is displayed across the top of the table for navigation and data modification. The Data page looks like this:



You can use the Database Pilot to view, edit, insert, and delete data in tables. The following procedures provide the steps needed to accomplish each task.

View table data

To view data in a table,

- 1 Select a table to view in the left pane.
- 2 Click the Data page tab in the right pane to view a scrollable table of all data in the table.
- 3 Use the toolbar buttons at the top of the table to scroll from record to record.

Edit a record

To edit a record,

- 1 Make sure that Request Updatable Queries on the Query page of the Options dialog box (View|Options) is checked.
- 2 Edit the record's fields in the table.
- 3 To post the edits to the local data set, select a different record in the table, or click the Post button in the Data page toolbar.
- 4 To cancel an edit before moving to another record, click the Cancel button in the toolbar or press *Esc*.
- 5 To save your changes to the database, click the Save Changes button.

Insert a new record

To insert a new record,

- 1 Select the row before which you want to insert a new row.
- 2 Click the Insert button on the Data page toolbar.

A blank row appears.

- 3 Enter data for each column.
Move between columns with the mouse, or by tabbing to the next field.
- 4 To post the insert to the local data set, select a different record in the table, or click the Post button in the Data page toolbar.
- 5 To cancel an insert before moving to another record, click the Cancel button in the toolbar or press *Esc*.
- 6 To save an insert to the database, click the Save Changes button.

Delete a record

To delete a record,

- 1 Place the cursor on the row you want to delete.
- 2 Click the Delete button.
- 3 To save a deletion from the database, click the Save Changes button.

Using the Database Pilot for database administration tasks

This section provides an introduction to creating, populating, and deleting tables in a SQL-oriented manner. These tasks are usually reserved for a Database Administrator, but can easily be accomplished using JBuilder.

Creating the SQL data source

JBuilder is an application development environment in which you can create applications that access database data, but it does not include menu options for features that create SQL server tables. Typically, this is an operation reserved for a Database Administrator (DBA). However, creating tables can easily be done using SQL and the Database Pilot.

This topic is not intended to be a SQL language tutorial but to show you how you can use SQL statements in JBuilder. For more information about the SQL syntax, refer to any book on the subject. One commonly used reference is *A Guide to the SQL Standard* by C.J. Date.

Note On many systems, the DBA restricts table create rights to authorized users only. If you have any difficulties with creating a table, contact your DBA to verify whether your access rights are sufficient to perform such an operation.

To create a simple table, you must first set up a database connection URL. If you are unfamiliar with how to do this, follow these steps:

- 1 Choose Tools|Database Pilot.
- 2 From the Database Pilot, choose File|New, or right-click an existing URL and select New from the context menu.

The New URL dialog box opens.

- 3 Select a Driver from the drop-down list or enter the driver information.

For a discussion of the different types of drivers, see JDBC database drivers in the Database Pilot online help.

- 4 Browse to or enter the desired URL.

The Browse button will be enabled when a database driver that is recognized by JBuilder is selected in the Driver field.

- 5 Click OK to close the dialog box.
- 6 Specify the UserName and any other desired properties on the Definition page in the right pane.
- 7 Click the Apply button on the toolbar to apply the connection parameters.

Once a connection has been established, you can specify a SQL statement to run against the database. There are two ways to do this. The first way is through the Create Table dialog. To create a table called mytable using the Create Table dialog,

- 1 Choose File>Create Table in the Database Pilot.
- 2 Type `mytable` in the Table name field.
- 3 Click the Insert button.
- 4 Type `lastName` in the Column name column.
- 5 Select `VARCHAR` as the Data type column value.
- 6 Type `20` in the Precision column.
- 7 Click the Next row button.

A new row is created.

- 8 Type `firstName` in the Column name column.
- 9 Select `VARCHAR` as the Data type column value.
- 10 Type `20` in the Precision column.
- 11 Click the Next row button.

A new row is created.

- 12 Type `salary` in the Column name column.
- 13 Select `NUMERIC` as the Data type column value.
- 14 Type `10` in the Precision column.
- 15 Type `2` in the Scale column.
- 16 Click the Execute button.
- 17 Note that a SQL statement has been created for you in the SQL text area.
- 18 Click OK.

The table is created in the currently open data source.

The second way to create a table is to specify a `CREATE TABLE` SQL statement in the Enter SQL tab. For example, to create a table named mytable2 on the data source to which you are connected,

- 1 Click the Enter SQL tab in the Database Pilot.
- 2 Enter the following in the text area:

```
create table mytable2 (
  lastName char(20),
  firstName char(20),
  salary numeric(10,2) )
```

- 3 Click the Execute button.

These steps create an empty table which can be used in a query. Use the Database Pilot to verify that the table was created correctly. You should see:

- A list of tables in the data source, including the new table (MYTABLE) just created.
- A list of columns for the selected table. Select MYTABLE and the columns list displays FIRSTNAME, LASTNAME and SALARY.

Populating a SQL table with data using JBuilder

Once you've created an empty table, you can easily fill it with data using the Database Pilot (in this example), or by creating an application using JBuilder's visual design tools. Select the Data page to display the data in a selected table, view, or synonym. You can enter and edit records in a table on the Data page of the Database Pilot if the table permits write access, and if Request Updatable Queries is checked in the View Options dialog box. The Data page displays a table populated with the data from the selected table.

- 1 Follow the steps for [“Creating the SQL data source” on page 170](#).
- 2 Select the table you just created in the left window, then select the Data tab in the right window.

A table populated with the data from the selected table displays in the right pane. A toolbar control is displayed across the top of the table for navigation and data modification.

- 3 Use the Database Pilot to view, edit, insert, and delete records in tables.

See [“Using the Explorer to view and edit table data” on page 168](#) for more information on using the Database Pilot to view and modify tables.

Deleting tables in JBuilder

Now that you've created one or more test tables, you'll need to know how to clean up and remove all the test tables. Follow the steps for [“Creating the SQL data source” on page 170](#) but substitute the following SQL statement:

```
drop table mytable
```

You can verify the success of this operation by checking to see if the table still displays in the left window of the Database Pilot.

Monitoring database connections

JBuilder provides a JDBC monitoring class which can monitor JDBC traffic. JBuilder provides a user interface, invoked from Tools\JDBC Monitor, to work with this class at design time. For information on using this class at run time, see [“Using the JDBC Monitor in a running application” on page 174](#).

JDBC Monitor will monitor any JDBC driver (that is, any subclass of `java.sql.Driver`) while it is in use by JBuilder. The JDBC Monitor monitors all output directly from the JDBC driver.

About the JDBC Monitor

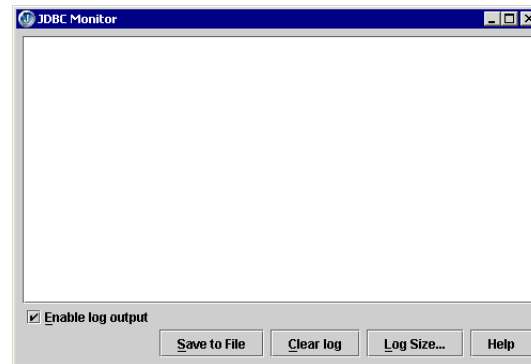
The typical use for the JDBC Monitor is to monitor JDBC traffic while you develop your database application.

To use the JDBC Monitor,

- 1 Choose Tools|JDBC Monitor.

The JDBC Monitor displays:

Figure 15.3 JDBC Monitor



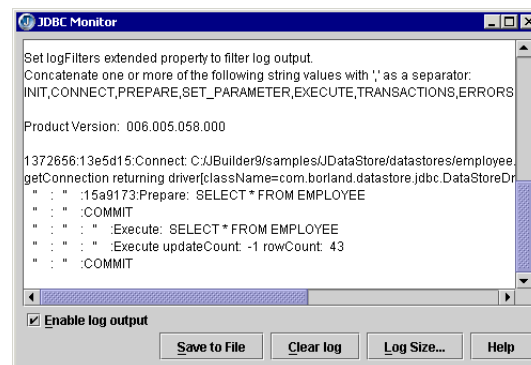
- 2 Open a source file that contains Data Access components that connect to a database, providing live data in the designer.

For example, most `Frame` classes in the DataExpress samples (`<jbuilder>/samples/DataExpress`) contain components that provide live data in design mode.

- 3 Click the Design tab to activate the designer.

Note how the JDBC Monitor displays database as the designer opens.

Figure 15.4 JDBC Monitor with output



You can perform the following actions in the JDBC Monitor:

- Click the JDBC Monitor window's close button to close the JDBC Monitor.
- Select text in the log area by highlighting it with the mouse or keyboard.
- Click the Save To File button to save the selected text (or all text, if nothing has been selected) to a file.
- Click the Clear Log button to clear the selected text (or all the text, if nothing has been selected).
- Click the Enable Log Output check box to enable/disable log output.
- Click the Log Size button to set the maximum amount of logging information to keep (8K by default).
- With the cursor in the text area, press *F1* or the Help button to display JDBC Monitor help. Help is available in design mode only.

Using the JDBC Monitor in a running application

To monitor database connections at run time, a `MonitorButton` or a `MonitorPanel` must be included with the application. `MonitorButton` is a Java bean which allows you to run the JDBC monitor against a running application. To do so, the instance of the JDBC monitor in use must be brought up by the application. An instance of the JDBC Monitor brought up from the IDE will only monitor database activities during design time. Clicking the Monitor button displays a dialog containing the JDBC Monitor.

The `MonitorPanel` can be used to place the monitor directly on a form. It has the same properties as the `MonitorButton`.

Adding the MonitorButton to the Palette

The `MonitorButton` can be put on the component palette by following these steps:

- 1 Choose Tools|Configure Palette to open the Palette Properties dialog box.
- 2 Select the Pages tab, and select DataExpress in the left pane.
- 3 Select the Add Components tab, and click Select Library to open the Select A Different Library dialog box.
- 4 Select JBCL from the list of available libraries, and click OK.
- 5 If the Palette Page To Receive Components drop-down list does not display DataExpress, select DataExpress from the drop-down list.

This determines the palette page on which the `MonitorPanel` button will be placed.

- 6 Select the No Filtering from the Component Filtering radio buttons.
- 7 Click Add From Selected Library to open the Browse For Class dialog box.
- 8 Browse to `com.borland.jbcl.sql.monitor.MonitorButton`, and click OK to add the component.
- 9 Click OK to close the Results dialog box.
- 10 Click OK to close the Palette Properties dialog box.

Using the MonitorButton Class from code

When the `MonitorButton` is added to the palette, it can be dropped on to your application. You could also add an instance of the `MonitorButton` in code, as follows:

```
MonitorButton monitorButton1 = new com.borland.jbcl.sql.monitor.MonitorButton();
this.add(monitorButton1);
```

Understanding MonitorButton properties

The following component properties are available on `MonitorButton` to control the default state of the monitor:

Property	Effect
<code>outputEnabled</code>	Turns Driver trace on/off.
<code>maxLogSize</code>	Maximum trace log size. Default is 8K.

Chapter 16

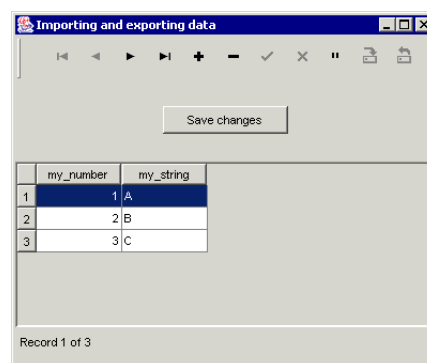
Tutorial: Importing and exporting data from a text file

This tutorial shows how to provide data to an application using a `TableDataSet` component and a text data file. For the tutorial, you will manually create the text data file, but this type of file can be exported from most desktop databases. For this tutorial, you will perform the following tasks:

- Create a JBuilder project
- Create a simple text data file
- Generate an application
- Add DataExpress components to access and store data from the text file
- Add dbSwing components to create a user interface
- Add a JButton Swing component for exporting data
- Compile and run your application
- Use patterns to export numeric, date/time, and text fields

When you have completed the tutorial, your application will look like this:

Figure 16.1 Import/export database application



The completed application can be viewed by opening the sample project file, `TextFileImportExport.jpx`, in `<jbuilder>/samples/DataExpress/TextFileImportExport/`. For users with read-only access to JBuilder samples, copy the `samples` directory into a directory with read/write permissions.

See the Tutorials section in the JBuilder Quick Tips for useful information about viewing and printing tutorials. The Accessibility options section in the JBuilder Quick Tips contains tips on using JBuilder features to improve JBuilder's ease of use for people with disabilities.

For information on documentation conventions used in this tutorial and other JBuilder documentation, see ["Documentation conventions" on page 6](#).

Step 1: Creating the project

To develop your database application in JBuilder, you first need to create a new project. To do this,

- 1 Choose **File/New Project** to display the Project wizard.
- 2 Type `TextFileImportExport` in the Name field.
- 3 Make sure the **Generate Project Notes File** option is checked.
- 4 Click **Finish** to close the Project wizard and create the project. You do not need to make any changes to the defaults on Steps 2 and 3 of the wizard.

The project file `TextFileImportExport.jpx` and project's HTML file are displayed in the project pane.

Step 2: Creating the text file

Now we will create a simple text data file for importing data into your database application. Create a text file as described in the following steps:

- 1 Create a new text file named `ImportTest.txt` in the directory containing the project file, `TextFileImportExport.jpx`, you created in the previous step.

Choose **File/New File** to open the Create New File dialog box. In the Name field, type `ImportTest`. Select `txt` from the Type drop-down list. Ensure that the Directory field is set to the directory containing `TextFileImportExport.jpx`, and the **Add Saved File To Project** check box is checked.

- 2 Enter the following three rows and two columns of data (a column of integer values and a column of string values) into the new text file.

Press *Enter* at the end of each row. Enter the quotation marks as well as the data.

```
1, "A"  
2, "B"  
3, "C"
```

- 3 Save and close the file.

Step 3: Generating an application

The Application wizard creates Java source files that are added to the project you just created.

To generate source files for your application using the Application wizard, follow these steps:

- 1 Choose File|New to open the object gallery.
- 2 Select General in the tree, and double-click the Application icon to open the Application wizard.
- 3 Accept the defaults in Step 1 of the Application wizard, and click Finish.

The new Java source files are added to your project and displayed as nodes in the project pane. The source code for `Frame1.java` is open in the content pane.

- 4 Choose File|Save All to save the source files and the project file.

Step 4: Adding DataExpress components to your application

The UI designer is used to add the DataExpress components to `Frame1.java`.

- 1 Select the Design tab for `Frame1.java` in the content pane.



- 2 Click the `TextDataFile` component on the DataExpress page of the component palette, and click in the component tree or the UI designer to add the component to your application.

The new `TextDataFile` component, `textDataFile1`, appears as a node in the component tree in the structure pane.

- 3 Select the following properties in the Inspector, and set their values as indicated:

To set the `fileName` property, select the field to the right of the property name and click the ellipsis (...) button to open the `FileName` dialog box. Click the ellipsis (...) button in the `FileName` dialog box and navigate to the `ImportTest.txt` file, select it, and click Open. Click OK to close the `FileName` dialog box.

Property name	Value
<code>delimiter</code>	" (double quote)
<code>separator</code>	, (comma)
<code>fileName</code>	<path_to_text_data_file> (the path to <code>ImportTest.txt</code> , including the file name)

A delimiter in a text file is a character that is used to define the beginning and end of a string field. By default, the *delimiter* for string data types is a double quotation mark. For this tutorial, no changes are needed.

A separator in a text file is a character that is used for differentiating between column values. By default, the *separator* character is a tab (`/t`). For this example, the separator is a comma (`,`). When using other text files, modify these properties accordingly.

Specify the complete path and file name for the `fileName` field.



- 4 Click the `TableDataSet` component on the DataExpress page of the component palette, and click in the component tree or UI designer to add the component to your application.
- 5 Select its `dataFile` property, and set it to `textDataFile1`.
- 6 Add columns to the `TableDataSet` component.

This tutorial describes adding columns to the data set through the UI designer. To add columns using the editor, see [“Adding columns to a TableDataSet in the editor” on page 23](#). If you have completed this tutorial previously and exported the data to a text file, JBuilder created a SCHEMA (.schema) file that provides column definitions when the text file is opened and you do not need to add columns manually.

- a Click the expand icon to the left of the `TableDataSet` component to expose existing columns.

In this case, there are no existing columns.

- b Select `<new column>`, and set the following properties in the Inspector for the first column:

Property name	Value
<code>dataType</code>	<code>SHORT</code>
<code>caption</code>	<code>my_number</code>
<code>columnName</code>	<code>my_number</code>

- c Select `<new column>` again, and set the following properties in the Inspector for the second column:

Property name	Value
<code>dataType</code>	<code>STRING</code>
<code>caption</code>	<code>my_string</code>
<code>columnName</code>	<code>my_string</code>

- 7 Choose File|Save All to save the source files and the project file.

You now have the basic components in place for retrieving and storing data from your text file. Next, you will create a user interface for displaying and editing the data.

Step 5: Adding dbSwing components to create a user interface

Now you are ready to create a user interface for your database application. The fastest way to do this is to use the dbSwing components in the UI designer.

Note Normally the first step in setting up a user interface is to determine the appropriate layout for your application (how the components are arranged visually, and which Java Layout Manager to use to control their placement.) However, learning how to use Java layout managers is a big task in itself. Therefore, to keep this tutorial focused on creating a database application, you'll use the default layout (`BorderLayout`), and control the placement of the components by setting their `constraints` property.

To learn about using layouts, see “Using the Designer,” and “Using layout managers” in *Designing Applications with JBuilder*.

The steps below add the following UI components to the application from the dbSwing tab on the component palette:

- **JdbTable** (and container), used to display two-dimensional data, in a format similar to a spreadsheet.
- **JdbNavToolBar**, a set of buttons that help you navigate through the data displayed in a **JdbTable**. It enables you to move quickly through the data set when the application is running.
- **JdbStatusLabel**, which displays information about the current record or current operation, and any error messages.

You will add these components to `contentPane` (`BorderLayout`), which is a `JPanel`, and the main UI container into which you are going to assemble the visual components.

- 1 Select the dbSwing page on the component palette in the UI designer.



- 2 Click the **JdbNavToolBar** component and click the area close to the center, top edge of the panel in the UI designer.

An instance of **JdbNavToolBar**, called `jdbNavToolBar1`, is added to the panel and is displayed in the component tree. By default, the **JdbNavToolBar** component automatically detects other data-aware components in the same root container (such as `JFrame`), and navigates the `DataSet` of the component that currently has focus. Therefore, you do not need to set the `dataSet` property for `jdbNavToolBar1` in the Inspector.

`jdbNavToolBar1` is now the currently selected component, and should extend across the top edge of the panel. Don't worry if it went somewhere different than you expected. The layout manager controls the placement, guessing the location by where you clicked. If you were too close to the left or right or middle of the panel, it may have guessed you wanted the component in a different place than you intended. You can fix that in the next step.

- 3 Look at the `constraints` property for `jdbNavToolBar1` in the Inspector.

It should have a value of `NORTH`. If it doesn't, click once on the value to display a drop-down list, and select `North` from the list.



- 4 Click the **JdbStatusLabel** component, and click the area close to the center, bottom edge of the panel in the UI designer.

An instance of **JdbStatusLabel**, called `jdbStatusLabel1`, is added to the panel and is displayed in the component tree. `jdbStatusLabel1` should have a `constraints` property value of `SOUTH`. If it doesn't, change it in the Inspector. `jdbStatusLabel1` automatically attaches itself to whichever `DataSet` has focus.



- 5 Click the **TableScrollPane** component on the dbSwing page of the component palette, and click in the center of the panel in the UI designer to add the component to your application.

The **TableScrollPane** component, `tableScrollPane1`, appears as a node in the component tree in the structure pane.



- 6 Click the **JdbTable** component on the dbSwing page of the component palette, and click in the component tree or UI designer to add the component to your application.

The **JdbTable** component, `jdbTable1`, appears as a node under `tableScrollPane1` in the component tree in the structure pane.

- 7 Set the `dataSet` property of `jdbTable1` to `tableDataSet1`.

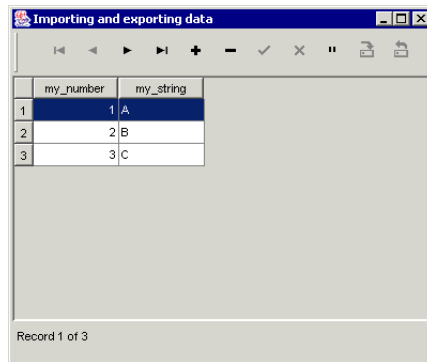
When you set the `dataSet` property of `jdbTable1` to `tableDataSet1`, the data from the text file appears in the UI designer:

You will see an error dialog if a valid data file is not specified or if the columns are not defined correctly. If you do not instantiate a visual component to view data, you must explicitly open the file in the source code to have access to the data.

- 8 Choose Run/Run Project to compile and run the application.

The running application looks like this:

Figure 16.2 Import/Export application at runtime



- 9 Close the running application.

When you run this application, the data in the text file is loaded into a `TableDataSet` and displayed in the visual table component to which it is bound. You can now view, edit, add, and delete data from the data set. A `TableDataSet` component can be used as either a master or a detail table in a master-detail relationship. To save changes back to the text file, you must export the data back. See [“Exporting data” on page 25](#) for more information on exporting.

Step 6: Adding a JButton Swing component

The JButton Swing component will be used to export data from the running application. Exporting data, or *saving data to a text file*, saves all of the data in the current view to the text file, overwriting the existing data. In this tutorial, you will resolve data from a `TableDataSet` to the text file used originally to import the data. When data is *exported* to a text file, all of the data in the current view is written to the text file, and the row status information is not affected.

- 1 Select the Design tab of the content pane.
- 2 Select `contentPane` (`BorderLayout`) in the content pane and change its `layout` property to `null` in the Inspector.
- 3 Select `tableScrollPane1` in the component tree, and in the UI designer, grab the upper handle and resize the component to allow room to add a button.

See the screen shot of the running application further in this tutorial for general placement of components.

- 4 Add a `JButton` component from the Swing page to the UI designer, and on the Properties tab of the Inspector, set the `text` property for the `JButton` component to `Save Changes`.
- 5 Click the Events tab of the Inspector, and select, then double-click the `actionPerformed()` method.

This changes the focus of the IDE from the Design tab to the Source tab, and displays the stub for the `actionPerformed()` method in the source code.

Add the following code to the `actionPerformed()` method:

```
try {
    tableDataSet1.getDataFile().save(tableDataSet1);
    System.out.println("Changes saved");
}
catch (Exception ex) {
    System.out.println("Changes NOT saved");
    System.err.println("Exception: " + ex);
}
```

- 6 Choose `File|Save All` to save the source files and the project file.

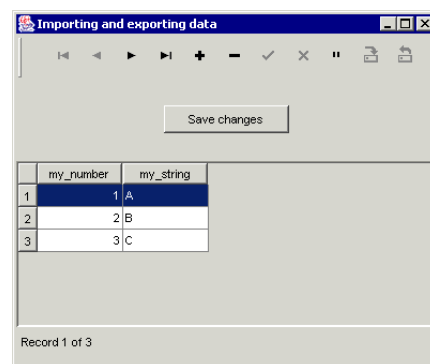
Step 7: Compiling and running your application

When you compile and run your application, it includes a `Save Changes` button for exporting data. When you export data from a `TableDataSet` to a text file, JBuilder creates a `SCHEMA (.schema)` file that defines the columns by name and data type. The next time you import the data into JBuilder, you do not have to define the columns, because this information is already specified in the `SCHEMA` file.

- 1 Run the application by choosing `Run|Run Project`.

When you run the application, the application appears in its own window. Data is displayed in a table, with a `Save Changes` button.

Figure 16.3 Exporting data to text file application at runtime



- 2 With the application running, select the string field in the first record of the `Frame` window and change the value in the field from `A` to `Apple`.
- 3 Save the changes back to the text file by clicking the `Save Changes` button.
- 4 Open `ImportTest.txt` in the content pane, and note that it now contains the following data:

```
1,"Apple"
2,"B"
3,"C"
```

5 Close the text file.

JBuilder automatically creates a SCHEMA file to define the contents of the text file.

6 View the SCHEMA file in a text editor. Notice that this file contains information about the name of the fields that have been exported and the type of data that was exported in that field. It looks like this:

```
[ ]
FILETYPE = VARYING
FILEFORMAT = Encoded
ENCODING = Cp1252
DELIMITER = "
SEPARATOR = ,
FIELD0 = my_number,Variant.SHORT,-1,-1,
FIELD1 = my_string,Variant.STRING,-1,-1,
```

7 Close the SCHEMA file.

You can continue to edit, insert, delete, and save data until you close the application, but you must click the Save Changes button to write any changes back to the text file. When you save the changes, the existing file will be overwritten with data from the current view.

Step 8: Using patterns for exporting numeric, date/time, and text fields

By default, JBuilder expects data entry and exports data of date, time, and currency fields according to the `locale` property of the column. You can use the `exportDisplayMask` property to read or save date, time, and number fields in a different pattern. The following steps demonstrate how to create an `exportDisplayMask` for a new column of type DATE.

1 Select `Frame1.java` in the content pane, then select the Design tab. Expand `tableDataSet1` in the component tree by clicking on the expand icon to its left. Select `<new column>`, then modify the column's properties in the Inspector as follows:

- `dataType` to DATE
- `caption` and `columnName` to `my_date`

2 Run the application. In the running application window, enter a date in the `my_date` column of the first row. By default, you must enter the date in a format of dd/MM/yy, like 16/11/95. Click the Save Changes button to save the changes back to the text file.

3 View the text file in a text editor. It will now contain the following data:

```
1,"Apple",1995-11-16
2,"B"
3,"C"
```

4 Close the text file.

5 View the SCHEMA file in a text editor. Notice that the new date field has been added to the list of fields. It looks like this:

```
[ ]
FILETYPE = VARYING
FILEFORMAT = Encoded
ENCODING = Cp1252
DELIMITER = "
SEPARATOR = ,
FIELD0 = my_number,Variant.SHORT,-1,-1,
FIELD1 = my_string,Variant.STRING,-1,-1,
FIELD2 = my_date,Variant.DATE,-1,-1,
```

6 Close the SCHEMA file.

The next steps show what happens when you change the date pattern, edit the data, and save the changes again.

- 1 Close the running application and the text files and return to the JBuilder UI designer. Select the `my_date` column and enter the following pattern into the `exportDisplayMask` property in the Inspector: `MM-dd-yyyy`. The syntax of patterns is defined in “String-based patterns (masks)” in the *DataExpress Component Library Reference*. This type of pattern will save the date field as follows: 11-16-1995.
- 2 The application would produce an error now if you tried to run it, because the format of the date field in the text file does not match the format the application is trying to open. Manually edit the text file and remove the value “,11/16/95” from the first row.

Instead of the above step, you could manually enter code that would establish one `exportDisplayMask` for importing the data and another `exportDisplayMask` for exporting the data.
- 3 Run the application, and enter a date, such as 16/11/1995, in the `my_date` column of the first row, and click the Save Changes button to save the changes back to the text file.
- 4 View the text file in a text editor. It will now contain the following data:

```
1,"Apple",11-16-1995
2,"B"
3,"C"
```

- 5 Close the text file.
- 6 View the SCHEMA file in a text editor. Notice that the date field format is displayed as part of the field definition. When the default format is used, this value is blank, as it is in the `FIELD0` definition. It looks like this:

```
[ ]
FILETYPE = VARYING
FILEFORMAT = Encoded
ENCODING = Cp1252
DELIMITER = "
SEPARATOR = ,
FIELD0 = my_number,Variant.SHORT,-1,-1,
FIELD1 = my_string,Variant.STRING,-1,-1,
FIELD2 = my_date,Variant.DATE,-1,-1,MM-dd-yyyy
```

- 7 Close the SCHEMA file.

When the text data file is imported next, the data will be imported from the information in the SCHEMA file. To view data in the table in a different pattern, set the `displayMask` property. To modify data in the table using a different pattern, set the `editMask` property. These properties affect viewing and editing of the data only; they do not affect the way data is saved. For example, to enter data into a currency field without having to enter the currency symbol each time, use a `displayMask` that uses the currency symbol, and an `editMask` that does not contain a currency symbol. You can choose to save the data back to the text file with or without the currency symbol by setting the `exportDisplayMask`.

Chapter 17

Tutorial: Creating a basic database application

This tutorial describes how to develop a sample database application using DataExpress components and the JBuilder design tools. Where necessary, the code generated by the design tools will be modified to provide custom behavior.

This application demonstrates the following functionality:

- Connects to the JDataStore sample database, `employee.jds`, using the `Database` and `QueryDataSet` components. (See [Chapter 4, “Connecting to a database”](#) and [“Querying a database” on page 46.](#))
- Contains a `JdbTable` which displays the data while also demonstrating the following features:
 - Persistent columns, which are columns where structure information typically obtained from the server is specified as a column property instead. This offers performance benefits as well as persistence of column-level properties. (See [“Persistent columns” on page 73](#) for more information on this feature.) In the designer, double-click the data set to open the column designer to view more information on each column.
 - Formatting of the data displayed in the `JdbTable` using display masks (the `HIRE_DATE` column). (See [“Adding an Edit or Display Pattern for data formatting” on page 148.](#))
 - Data editing that is controlled using edit masks (the `HIRE_DATE` column). (See [“Adding an Edit or Display Pattern for data formatting” on page 148.](#))
 - Calculated and aggregated fields which get their values as a result of an expression evaluation (the `NEW_SALARY`, `ORG_TOTAL`, `NEW_TOTAL`, `DIFF_SALARY`, AND `DIFF_TOTAL` columns). (See [“Using calculated columns” on page 142.](#))

- Includes a `JdbStatusLabel` control that displays navigation information, data validation messages, and so on. Messages are written to the `JdbStatusLabel` control when appropriate, or when instructed programmatically. (See [“Displaying status information” on page 158.](#))
- Displays a `JdbNavToolBar` for easy navigation through the data displayed in the table.
- Lets you locate data interactively using a `JdbNavField` which is embedded in the `JdbNavToolBar`. For more information on locating data, see [“Locating data” on page 130.](#)
- Uses a `DBDisposeMonitor` to automatically close the database connection when the frame is closed.
- Resolves changes made to the data in the `QueryDataSet` by using default resolver behavior. (See [“Understanding default resolving” on page 87.](#)) The Save button of the `JdbNavToolBar` performs the save. Messages regarding the resolve process are displayed in the `JdbStatusLabel` control.

For this tutorial, you will perform the following tasks:

- Create a JBuilder project
- Generate an application
- Add DataExpress components to access data from the database
- Design the columns for the application
- Add dbSwing components to create a user interface
- Aggregate data with calculated fields

When you have completed the tutorial, your application will look like this:

Figure 17.1 Basic database application



The screenshot shows a window titled "Sample Database Application". It features a table with columns: Employee No, Name, Hire Date, D..., Country, and Salary. The table contains 20 rows of employee data. Below the table, there are summary fields: Original Total (115,522,467.99), New Total (127,074,714.79), and Difference (11,552,246.8). A status bar at the bottom indicates "Record 1 of 42".

	Employee No	Name	Hire Date	D...	Country	Salary
1	2	Nelson, Robert	12-28-88	600	USA	105,900
2	4	Young, Bruce	12-28-88	621	USA	97,500
3	5	Lambert, Kim	02-06-89	130	USA	102,750
4	8	Johnson, John	04-05-89	180	USA	64,635
5	9	Forest, Phil	04-17-89	622	USA	75,060
6	11	Weston, K. J.	01-17-90	130	USA	86,292.94
7	12	Lee, Terri	05-01-90	000	USA	53,793
8	14	Hall, Stewart	06-04-90	900	USA	69,482.62
9	15	Young, Katherine	06-14-90	623	USA	67,241.25
10	20	Papadopoulos, Chris	01-01-90	671	USA	89,655
11	24	Fisher, Pete	09-12-90	671	USA	81,810.19
12	28	Bennet, Ann	02-01-91	120	England	22,935
13	29	De Souza, Roger	02-18-91	623	USA	69,482.62
14	34	Baldwin, Janet	03-21-91	110	USA	61,637.81
15	36	Reeves, Roger	04-25-91	120	England	33,620.62
16	37	Stansbury, Willie	04-25-91	120	England	39,224.06
17	44	Phong, Leslie	06-03-91	623	USA	56,034.38
18	45	Ramanathan, Ashok	08-01-91	621	USA	80,689.5
19	46	Steadman, Walter	08-09-91	900	USA	116,100
20	52	Nordstrom, Carol	10-02-91	180	USA	42,742.5

Original Total 115,522,467.99 New Total 127,074,714.79 Difference 11,552,246.8

Record 1 of 42

The completed application can be viewed by opening the sample project file, `BasicApp.jpx`, in `<jbuilder>/samples/DataExpress/BasicApp/`. There may be minor differences between the application created in this tutorial and the sample application. For users with read-only access to JBuilder samples, copy the `samples` directory into a directory with read/write permissions.

See the Tutorials section in the JBuilder Quick Tips for useful information about viewing and printing tutorials. The Accessibility options section in the JBuilder Quick Tips contains tips on using JBuilder features to improve JBuilder's ease of use for people with disabilities.

For information on documentation conventions used in this tutorial and other JBuilder documentation, see [“Documentation conventions” on page 6](#).

Step 1: Creating the project

To develop your database application in JBuilder, you first need to create a new project. To do this,

- 1 Choose **File|New Project** to display the Project wizard.
- 2 Type `BasicApp` in the Name field.
- 3 Click **Finish** to close the Project wizard and create the project. You do not need to make any changes to the defaults on Steps 2, 3, or 4 of the wizard.

The project file `BasicApp.jpx` is displayed in the project pane.

Step 2: Generating an application

The Application wizard creates `.java` source files that are added to the project you just created.

To generate source files for your application using the Application wizard, follow these steps:

- 1 Choose **File|New|General** to open the General panel in the object gallery.
- 2 Double-click the Application icon to open the Application wizard.
- 3 In Step 1 of the Application wizard, accept the default package name, `basicapp`, type `BasicApp` in the Class Name field, and click **Next**.

Note

The package name used in this tutorial, `basicapp`, differs from the package name used in the sample application, `com.borland.samples.dx.basicapp`, but the applications are otherwise the same.

- 4 In Step 2 of the Application wizard, type `BasicAppFrame` in the Class field, type `Sample Database Application` in the Title field, and click **Finish**.

The new Java source files are added to your project and displayed as nodes in the project pane. The source code for `BasicAppFrame.java` is open in the content pane.

- 5 Choose **File|Save All** to save the source files and the project file.

Step 3: Adding DataExpress components to your application

The UI designer is used to add the DataExpress components to `BasicAppFrame.java`. You will add the following DataExpress components to your application:

- Database
- QueryDataSet
- DBDisposeMonitor

These components provide the underlying database framework for the application.

- 1 Select the Design tab for `BasicAppFrame.java` in the content pane to activate the UI designer.

The component palette appears on the side of the UI designer.



- 2 Select the DataExpress page of the component palette, click the `Database` component, and then click in the structure pane or the UI designer to add the component to the application.

The new `Database` component, `database1`, shows up under the Data Access node in the structure pane, and the following line of code is added to the Frame class:

```
Database database1 = new Database();
```

- 3 Select the `database1` component in the structure pane, select the `connection` property in the Inspector, and click the ellipsis (...) button to open the Connection dialog box.
- 4 Set the connection properties to the JDataStore sample employee table, using the field values in the following table.

The Connection URL points to the `employee.jds` file in a subdirectory of your JBuilder installation directory, `<jbuilder>`.

Property name	Value
Driver	<code>com.borland.datastore.jdbc.DataStoreDriver</code>
URL	Browse to your copy of <code><jbuilder>/samples/JDataStore/datastores/employee.jds</code>
Username	Enter your name (the default is "SYSDBA")
Password	Enter your password (the default is "masterkey")

The Connection dialog box includes a Test Connection button. Click this button to check that the connection properties have been correctly set. Results of the connection attempt are displayed in the status area. When the connection is successful, click OK. If the connection is not successful, make sure you have followed all the steps for [Chapter 4, "Connecting to a database."](#)



- 5 Select the DataExpress page of the component palette, click the `QueryDataSet` component, and then click in the structure pane or the UI Designer to add the component to the application.

The new `QueryDataSet` component, `queryDataSet1`, shows up under the Data Access node in the structure pane, and the following line of code is added to the Frame class:

```
QueryDataSet queryDataSet1 = new QueryDataSet();
```

- 6 Select the `query` property of the `QueryDataSet` component in the Inspector, click the ellipsis (...) button to open the Query dialog box, and set the following properties:

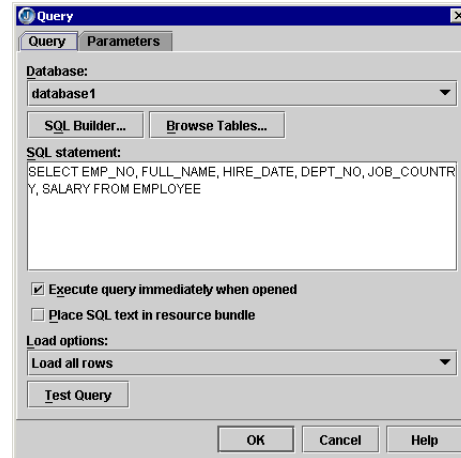
Property name	Value
Database	<code>database1</code>
SQL Statement	<code>SELECT EMP_NO, FULL_NAME, HIRE_DATE, DEPT_NO, JOB_COUNTRY, SALARY FROM EMPLOYEE</code>

The SQL Statement will be run against the specified Database automatically when the `QueryDataSet` is opened.

- 7 Click Test Query to ensure that the query is runnable.

The Query dialog box should look like this to indicate the query was successful.

Figure 17.2 Query dialog box



If the Query dialog box indicates Fail, review the information you have entered in the query for spelling and omission errors.

- 8 Click OK to close the Query dialog box.
- 9 Select the More dbSwing page of the component palette, click the `DBDisposeMonitor` component, and click in the structure pane to add the component to your application.

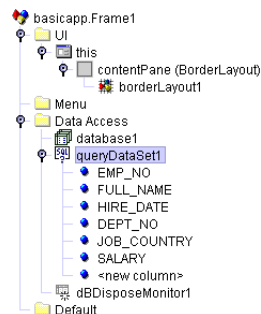
The new `DBDisposeMonitor` component, `dbDisposeMonitor1`, shows up under the Data Access node in the structure pane, and the following line of code is added to the Frame class:

```
DBDisposeMonitor dbDisposeMonitor1 = new DBDisposeMonitor();
```

The `DBDisposeMonitor` will close the `JDataStore` when the window is closed.

- 10 Set the `dataAwareComponentContainer` property for the `DBDisposeMonitor` to this.
- 11 Expand the `queryDataSet1` node in the structure pane.

Figure 17.3 queryDataSet1 node expanded



The selected columns of the sample `JDataStore` Employee database, `employee.jds`, are displayed in the `queryDataSet1` node.

You now have the basic components in place for retrieving and storing data from the Employee database. Next, you will create a user interface for displaying and editing the data.

Step 4: Designing the columns for the application

Before we add a user interface to the application, we will,

- Add new columns and edit existing columns
- Specify a calculation for the calculated columns

Adding columns and editing column properties

- 1 Expand the `queryDataSet1` node in the structure pane, and double-click `<new column>` to add a new column.

This opens the column designer in the content pane, and loads the properties for the new column in the inspector.

- 2 Change the `columnName` property in the Inspector from `NewColumn1` to `NEW_SALARY`.



- 3 Click the Insert Column button to add additional four additional columns with the following `columnName` property values:

- `DIFF_SALARY`
- `ORIG_TOTAL`
- `NEW_TOTAL`
- `DIFF_TOTAL`

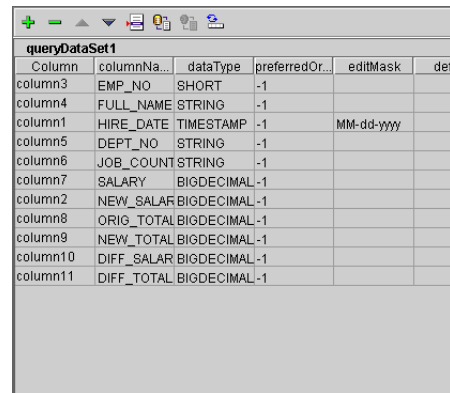
- 4 Set properties for the columns as described in the following table:

Column	Property name	Value
HIRE_DATE	caption	Hire Date
HIRE_DATE	displayMask	MM-dd-yy
HIRE_DATE	editMask	MM-dd-yyyy
NEW_SALARY	caption	NEW_SALARY
NEW_SALARY	calcType	calculated
NEW_SALARY	dataType	BIGDECIMAL
NEW_SALARY	visible	FALSE
EMP_NO	caption	Employee No
FULL_NAME	caption	Name
FULL_NAME	width	16
DEPT_NO	caption	Dept.
JOB_COUNTRY	caption	Country
JOB_COUNTRY	width	15
SALARY	caption	Salary
ORIG_TOTAL	calcType	aggregated
ORIG_TOTAL	caption	ORIG_TOTAL
ORIG_TOTAL	dataType	BIGDECIMAL
NEW_TOTAL	calcType	aggregated
NEW_TOTAL	caption	New Total
NEW_TOTAL	dataType	BIGDECIMAL
DIFF_SALARY	calcType	calculated
DIFF_SALARY	caption	DIFF_SALARY
DIFF_SALARY	dataType	BIGDECIMAL
DIFF_TOTAL	calcType	aggregated
DIFF_TOTAL	caption	Diff. Total
DIFF_TOTAL	dataType	BIGDECIMAL
DIFF_TOTAL	visible	FALSE

Editing the properties for a column makes the column persistent. When a column has become persistent, square brackets ([]) are placed around the column name in the structure pane. Optionally, you can set the `preferredOrder` property for the added columns to `-1` to accept the default column order.

When you're done editing the columns, the column designer should look similar to this:

Figure 17.4 queryDataSet1 columns in the column designer



Column	columnName	dataType	preferredOr...	editMask	def
column3	EMP_NO	SHORT	-1		
column4	FULL_NAME	STRING	-1		
column1	HIRE_DATE	TIMESTAMP	-1	MM-dd-yyyy	
column5	DEPT_NO	STRING	-1		
column6	JOB_COUNT	STRING	-1		
column7	SALARY	BIGDECIMAL	-1		
column2	NEW_SALARY	BIGDECIMAL	-1		
column8	ORIG_TOTAL	BIGDECIMAL	-1		
column9	NEW_TOTAL	BIGDECIMAL	-1		
column10	DIFF_SALARY	BIGDECIMAL	-1		
column11	DIFF_TOTAL	BIGDECIMAL	-1		

Specifying calculations for the calculated columns

The `NEW_SALARY` and `DIFF_SALARY` columns are calculated columns. In this tutorial, we are giving each employee a 10% raise. The calculation adds the existing `SALARY` data to the product of the existing `SALARY` data and 0.10. The resulting value is placed in the `NEW_SALARY` column. The `DIFF_SALARY` is calculated by subtracting the existing `SALARY` from the `NEW_SALARY`.

To add the calculation,

- 1 Select the `queryDataSet1` node in the structure pane, select the Events tab of the Inspector, and double-click the `calcFields` event handler.

This creates the stub for the event's method in `BasicAppFrame.java`, and displays the source code for the new method in the content pane.

- 2 Add the following statement to the existing import statements in `BasicAppFrame.java` to import the `java.math.BigDecimal` class needed for the `BIGDECIMAL` data type specified for the calculated columns.

```
import java.math.BigDecimal;
```

- 3 Modify the event method to calculate the value for `NEW_SALARY` and `DIFF_SALARY`, as follows:

```
void queryDataSet1_calcFields(ReadRow changedRow, DataRow calcRow, boolean isPosted)
    throws DataSetException {
    BigDecimal bDin = changedRow.getBigDecimal("Salary");
    BigDecimal bDout = bDin.add(new BigDecimal(bDin.doubleValue()*10.0/100));
    calcRow.setBigDecimal("NEW_SALARY", bDout);
    calcRow.setBigDecimal("DIFF_SALARY", bDout.subtract(bDin));
}
```

This method is called for `calcFields` whenever a field value is saved and whenever a row is posted. This event passes in an input which is the current values in the row (`changedRow`), an output row for putting any changes you want to make to the row (`calcRow`), and a boolean (`isPosted`) that indicates whether the row is posted in the `DataSet` or not. You may not want to recalculate fields on rows that are not posted yet.

- 4 Choose `File|Save All` to save the source files and the project file.

Some of the columns we added in this step are aggregated columns. We will address these columns later in the tutorial. Let's add a user interface to the application so we can see what the application looks like.

Step 5: Adding dbSwing components to create a user interface

Now you are ready to create a user interface for your database application. The fastest way to do this is to use the dbSwing components in the UI designer.

Note Normally the first step in setting up a user interface is to determine the appropriate layout for your application (how the components are arranged visually, and which Java Layout Manager to use to control their placement.) However, learning how to use Java layout managers is a big task in itself. Therefore, to keep this tutorial focused on creating a database application, you'll use the default layout (`BorderLayout`), and control the placement of the components by setting their `constraints` property.

To learn about using layouts, see "Introducing the Designer," and "Using layout managers" in *Designing Applications with JBuilder*.

The steps below add the following UI components to the application from the dbSwing page on the component palette:

- `JdbTable` (and container), used to display two-dimensional data, in a format similar to a spreadsheet.
- `JdbNavToolBar`, a set of buttons that help you navigate through the data displayed in a `JdbTable`. It enables you to move quickly through the data set when the application is running.
- `JdbStatusLabel`, which displays information about the current record or current operation, and any error messages.

You will add these components to `contentPane` (`BorderLayout`), which is a `JPanel`, and the main UI container into which you are going to assemble the visual components. Additional `JPanel` components will be used to separate the navigation components from the `JdbStatusLabel`.

To add the `JdbTable` component,

- 1 Select the Design tab for `BasicAppFrame.java` in the content pane to activate the UI designer.
- 2 Select the dbSwing page on the component palette in the UI designer.



- 3 Click the `TableScrollPane` component on the component palette, click `contentPane` in the component tree in the structure pane, or click in the center of the design surface in the UI designer to add the component to your application.

The `TableScrollPane` component, `tableScrollPane1`, appears as a node in the component tree in the structure pane.



- 4 Click the `JdbTable` component on the component palette, and click `tableScrollPane1` in the component tree in the structure pane, or click in the center of the design surface in the UI designer to add the component to your application.

The `JdbTable` component, `jdbTable1`, appears as a node under `tableScrollPane1` in the component tree in the structure pane.

- 5 Set the `dataSet` property of `jdbTable1` to `queryDataSet1`.

When you set the `dataSet` property of `jdbTable1` to `queryDataSet1`, the data from the database appears in the UI designer:

Figure 17.5 JdbTable component in the UI designer

	Employee No	Name	Hire Date	De...	J
1	2	Nelson, Robert	12-28-88	600	U
2	4	Young, Bruce	12-28-88	621	U
3	5	Lambert, Kim	02-06-89	130	U
4	8	Johnson, Leslie	04-05-89	180	U
5	9	Forest, Phil	04-17-89	622	U
6	11	Weston, K. J.	01-17-90	130	U
7	12	Lee, Terri	05-01-90	000	U
8	14	Hall, Stewart	06-04-90	900	U
9	15	Young, Katherine	06-14-90	623	U
10	20	Papadopoulos, Chris	01-01-90	671	U
11	24	Fisher, Pete	09-12-90	671	U
12	28	Bennet, Ann	02-01-91	120	E
13	29	De Souza, Roger	02-18-91	623	U

Next, we'll add some navigation components, including a `JdbNavToolBar` component. `JPanel` components will help separate the different types of UI elements.

To add the navigation elements,



- 1 Select the Swing Containers page on the component palette in the UI designer, click the `JPanel` component, and click the `contentPane` node in the structure pane.



- 2 Set the `layout` property for `jPanel1` to `FlowLayout`.

- 3 Select the More dbSwing page on the component palette, click the `JdbNavField` component, and click the `jPanel1` node in the structure pane.

The `JdbNavField` includes an incremental search feature for `String` type columns. Its `columnName` property specifies the column in which to perform the locate. If not set, the locate is performed on the last column visited in the `JdbTable`.

- 4 Set the `preferredSize` property for `jdbNavField1` to 125, 21.

label

- 5 Select the Swing page on the component palette, click the `JLabel` component, and click the `jPanel1` node in the structure pane.

- 6 Set the `text` property for `jLabel1` to Find.



- 7 Select the dbSwing page on the component palette, click the `JdbNavToolBar` component, and click the `jPanel1` node in the structure pane.

An instance of `JdbNavToolBar`, called `jdbNavToolBar1`, is added to the panel and is displayed in the component tree. By default, the `JdbNavToolBar` component automatically detects other data-aware components in the same root container, and navigates the `DataSet` of the component that currently has focus. Therefore, you do not need to set the `dataSet` property for `jdbNavToolBar1` in the Inspector.

Note

You may need to resize the designer workspace to see all the UI components.

Now, we're ready to add the `JdbStatusLabel` component.

To add the `JdbStatusLabel` component,

- 1 Add another `JPanel` component to the `contentPane` node in the structure pane.

- 2 In the Inspector, set the `constraints` property for `jPanel2` to `South`.



- 3 Select dbSwing page on the component palette, click the `JdbStatusLabel` component, and click the area close to the center, bottom edge of the panel in the UI designer.

An instance of `JdbStatusLabel`, called `jdbStatusLabel1`, is added to the panel and is displayed in the component tree. `jdbStatusLabel1` automatically attaches itself to whichever `DataSet` has focus.

- Choose Run/Run Project to compile and run the application.

The running application looks like this:

Figure 17.6 Basic database application with navigation bar and status label



Use the navigation bar and navigation field to move through the records. Note how the status bar display updates as you navigate.

- Close the running application, and save all changes (File/Save All).

To complete the application, let's add some `JdbTextField` components to the UI to display the data from the aggregated columns.

Step 6: Aggregating data with calculated fields

Now we'll add `JdbTextField` components to display the data from the aggregated columns, `ORIG_TOTAL`, `NEW_TOTAL`, and `DIFF_TOTAL`. These components will be contained in a separate `JPanel` component within the `JPanel` component that contains the `JdbStatusLabel`.

To add the `JdbTextField` components for the aggregated column data,

- Select the Swing Containers page on the component palette in the UI designer, select the `JPanel` component, and click the `jPanel12` node in the structure pane.

This adds a new `JPanel` component, `jPanel13`, within `jPanel12`.

- Set the layout property for `jPanel13` to `GridLayout` and the layout property for `jPanel12` to `BorderLayout`.



- Select the `dbSwing` page on the component palette, click the `JdbTextField` component, and click the `jPanel13` node in the structure pane.

- Set the `dataSet` property for `jdbTextField1` to `queryDataSet1`, set the `columnName` property to `ORIG_TOTAL`.

- Select the Swing page on the component palette, select the `JLabel` component, and click the `jPanel13` node in the structure pane.

If necessary, reposition the `JLabel` component (`jLabel12`) in the UI designer to put it at the left of the `jdbTextField1` component.

- Set the `horizontalAlignment` property for `jLabel12` to `LEADING`, and the `text` property to `Original Total`.

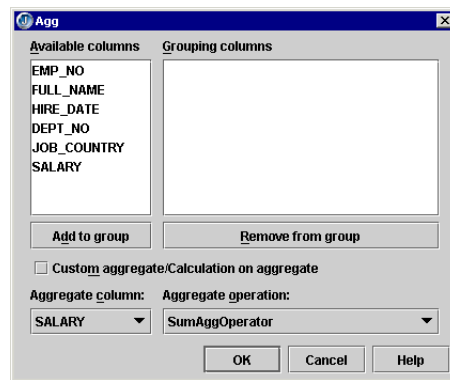
- 7 Add two more `JdbTextField` and `JLabel` components, and set their properties as described in the following table:

Component	Property name	Value
<code>jdbTextField2</code>	<code>dataSet</code>	<code>queryDataSet1</code>
<code>jdbTextField2</code>	<code>columnName</code>	<code>NEW_TOTAL</code>
<code>jLabel3</code>	<code>horizontalAlignment</code>	<code>CENTER</code>
<code>jLabel3</code>	<code>text</code>	<code>New Total</code>
<code>jdbTextField3</code>	<code>dataSet</code>	<code>queryDataSet1</code>
<code>jdbTextField3</code>	<code>columnName</code>	<code>DIFF_TOTAL</code>
<code>jLabel4</code>	<code>horizontalAlignment</code>	<code>CENTER</code>
<code>jLabel4</code>	<code>text</code>	<code>Difference</code>

If necessary, adjust the positioning of the components in the UI designer.

- 8 Expand the `queryDataSet1` node in the structure pane, and select the `ORIG_TOTAL` column.
- 9 In the Inspector, select the `agg` property, and click the ellipsis (...) button to open the Agg dialog box.

Figure 17.7 Agg dialog box



- 10 Choose `SALARY` from the Aggregate Column drop-down list, choose `SumAggOperator` from the Aggregate Operation drop-down, and click OK.
- 11 Select the `NEW_TOTAL` column in the structure pane, and open the Agg dialog box.
- 12 Choose `NEW_SALARY` from the Aggregate Column drop-down list, choose `SumAggOperator` from the Aggregate Operation drop-down, and click OK.
- 13 Select the `DIFF_TOTAL` column in the structure pane, and open the Agg dialog box.
- 14 Choose `DIFF_SALARY` from the Aggregate Column drop-down list, choose `SumAggOperator` from the Aggregate Operation drop-down, and click OK.
- 15 Choose Run/Run Project to compile and run the application.

The application should display the aggregated data in the new `JdbTextField` components.

Index

Symbols

? as JDBC parameter marker 57

A

accessing data 27, 45
 from custom data sources 65
 from data module 103
 from JDBC data sources 27
 from UI components 158
accessing model information 158
adding columns
 for internal purposes 75
 to imported text files 23
 to parameterized queries 57
adding components
 to data modules 104
adding parameters to queries 53
agg property editor 147
AggDescriptor objects 147
aggregating data 135
 creating aggregated columns 144, 147
 customizing aggregation methods 147
 sample 144, 185
applications
 database (2-tier) 109
 generating 117
Apply button 113
ascending sort order 114
ASCII files 23
 See also text files

B

binding parameter values 58
boolean data patterns 152
 examples of 150
boolean patterns 148
Borland
 contacting 7
 developer support 7
 e-mail 8
 newsgroups 8
 online resources 7
 reporting bugs 8
 technical support 7
 World Wide Web 8
bugs, reporting 8
building database applications 1
bundling resources 49
business logic 103, 106

C

calculated columns 135, 142, 144
 aggregating data 144, 147, 185
 creating lookups with 136, 138
 creating pick lists with 136
 example 143
 tutorial 185
 types supported 142
CalculatedColumn sample 143

calculating 147
 cost of goods 147
 discounts 147
 sales tax 147
 totals 147
cascadeDeletes option 98
cascadeUpdates option 98
client database applications, developing with
 InterClient 31
closing data sets 52
coding events
 for data modules 106
Column component 16, 69
 editing containing Java object 155
 formatter property 155
 formatting containing Java object 155
 locale property 175
 manipulating 154
 overview 16, 69
 persistent 153, 154
 setting properties 153, 154
 specifying as persistent 73
 storing Java objects in 155
 viewing 153
column designer 70, 153, 154
 enabling 70
 metadata options 71, 72
 RowIterator Generator button 71
column order 134
column properties
 data display 69
 for multiple table queries 85
columns 69, 73
 adding to StorageDataSets 75
 calculated 135
 changing properties for 70
 controlling order of 75
 exploring 165
 filtering data in 121
 linking on common 94
 locating data in 121
 lookup values in 136
 See also lookup columns
 setting persistent properties for 73
 setting properties for 69
 sorting 121
 viewing information 70, 72
 working with 69
common fields 93
components
 JFC data-aware 157
 synchronizing 157
.config files
 creating for drivers 33
connecting to a database
 tutorial 185
connections 27, 28
 overview 27
 pooling JDBC 38
 problems and solutions 43
 properties for databases 29
 tutorial 185

- constraints
 - enabling 129
- controlling user input 149
- controls 157
- Create ResourceBundle dialog box 49
- creating
 - master-detail relationships 93, 99
 - queries 46
 - SQL tables 170
- cursors
 - shared 157
- custom providers 65
- custom resolvers 88

D

- data 45
 - accessing 122
 - alternate view 152
 - caching 19
 - editing 168
 - exploring 172
 - exporting 25
 - extracting from a data source 45
 - filtering 121
 - finding 130
 - inserting 172
 - loading 67
 - locating 121, 130, 132, 133
 - manipulating 121
 - modifying 172
 - persisting 19, 153
 - providers 45
 - providing 45, 67
 - required 154
 - resolving 77, 78
 - customizing 90
 - resolving data
 - default behavior 87
 - resolving with stored procedures 80
 - retrieving 45, 65, 122
 - sorting 121
 - storing 19
 - viewing 168
- data constraints, enabling 129
- data fields, exporting 175
- data filters 124
 - example 124
- data groups 144
- data import and export
 - tutorial 175
- data input with patterns 135
- data members
 - nontransient 86
 - private 86
- Data Modeler
 - 2-tier applications 109
 - client-server applications 109
 - creating
 - queries 166
- Data Module wizard 104
- data modules 103
 - adding business logic 106
 - adding components 104
 - adding to libraries 106
 - class files 106
 - compiling 106
 - creating 104, 109
 - referencing 106, 108
 - saving 106
 - using 106, 108
 - using generated 118
 - wizard 104
 - wizards 108
- data patterns 148
 - examples of 150
- data providers 45
- data relationships 84
 - 1-to-1 84
 - 1-to-many 84
 - many-to-1 84
 - many-to-many 84
- data retrieval enhancements 51
- data sets 12, 52
 - binding parameter values 58
 - closing 52
 - enhancing performance 51
 - explicitly opening 52
 - linking 93
 - opening 67
 - returning as read-only 52
 - streamable 85
- data sources
 - accessing 45
 - connecting to 28
- data summaries 144, 185
- data tables 98
 - displaying detail link columns 98
- data types
 - variant 155
- data-aware components 157
 - default data display 69
 - displaying columns in 153
- database administration 165, 170
- database applications 11
 - creating 122
 - distributed 161
 - generating 117
 - introduction 1, 11
- Database class 14, 27, 28
 - example 28
 - overview 14, 27
 - using 28
- database connections 27
 - monitoring 172, 174
 - pooling 38
 - properties 29
- database drivers
 - adding to JBuilder 33
 - adding to project 34
 - all-Java 31
 - setting up 166
- database examples 87
 - alternate views 152
 - calculated aggregations 144
 - calculated columns 143
 - creating lookups 138
 - creating pick lists 136
 - filtering data 124
 - master-detail relationships 94
 - parameterizing queries 53
 - ResolverEvents 89

- resolving changes 78
 - resolving ProcedureDataSets 80
 - setting up JDataStore 29, 31
 - StreamableDataSets 162
 - viewing column information 70
 - Database Pilot 72
 - Data page 168, 172
 - Enter SQL page 167
 - setting up 166
 - using 165
 - viewing column information 72
 - window 166
 - database servers, communicating with 11
 - database tables 165
 - exploring 165
 - database tutorials 3
 - basic 185
 - calculated aggregations 185
 - calculated columns 175
 - data import and export 175
 - JDBC connections 185
 - numeric fields 175
 - text fields 175
 - text file 175
 - time fields 175
 - database URLs 165
 - database-related packages 14
 - databases 1, 28
 - accessing 166
 - connecting to 28
 - connecting via JDBC 78
 - connection property 29
 - developing 1
 - displaying information 166
 - exploring 165
 - in distributed applications 161
 - indexes 165
 - properties (examples) 135
 - querying 47
 - UI 121, 135
 - DataExpress 11
 - applications 11
 - architecture 1, 11, 12
 - components 12, 14, 45
 - DataExpress Component Library 11
 - DataExpress components
 - accessing data with 45
 - DataExpress for EJB components 17
 - DataModule interface 16
 - customizing 104
 - discussed 103
 - overview 16
 - referencing 108
 - DataRow component 16
 - column order in locates 134
 - locating data 132, 133
 - overview 16
 - DataSet component 14
 - filtering data in 121
 - functionality 12
 - locating data in 121
 - overview 14
 - saving changes 77
 - sorting data in 121
 - storing Java objects 155
 - streamable 85
 - with RMI 85
 - DataSet package 14
 - DataSetData component 85
 - example 161
 - extractDataSet method 85, 86
 - extractDataSetChanges method 86
 - passing metadata 163
 - populating 86
 - sample application 161
 - sample application, described 161
 - DataSetException class 159
 - DataSetView component 15
 - overview 15
 - properties 152
 - sorting in 127
 - using 135, 152
 - DataStore component 15
 - DataStoreDriver component 15
 - date data, patterns 150
 - dates, importing 24
 - DBA 170
 - tasks 165
 - dbSwing components 17
 - creating database UI 17
 - using 157
 - default project, adding database drivers to 34
 - Delay Fetch of Detail Records Until Needed 97
 - delete procedures, custom 81
 - deleteProcedure property 81
 - deleting
 - persistent columns 74
 - tables 172
 - deploying
 - multi-tier applications 164
 - descending sort order 114
 - designers
 - column designer 70
 - See also* column designer
 - detail records
 - fetching 97
 - detail tables 94
 - editing 98
 - Developer Support 7
 - display masks 135, 149
 - adding 148
 - displaying
 - data in data-aware components 69
 - special characters 149
 - status information 158
 - distributed applications
 - database 161
 - distributed objects
 - database 161
 - documentation conventions 6
 - platform conventions 7
 - driver manager 27
 - drivers
 - adding database drivers to project 34
 - adding JDBC 33
 - setting up for databases 166
- ## E
-
- edit masks 135
 - adding 148
 - editMask property 149

- edit/display masks 135
- editing data
 - controlling user input 149
 - master-detail 98
- enableDelete property 152
- enableInsert property 152
- enableUpdate property 152
- Enter SQL page (Database Pilot) 167
- errors
 - handling exceptions 159
 - See also* exceptions
- escape sequences 62
- event handlers
 - custom aggregation 147
- events
 - adding business logic 106
 - resolver 87
- examples
 - adding status information 158
 - database resolvers 87
 - filtering data 124
 - master-detail relationships 94
 - parameterizing queries 53
 - ResolverEvents 89
 - resolving data changes 78
 - stored procedures
 - coding 81
 - viewing column information 70
- exceptions
 - handling 159
- export masks 148, 149
- exportDisplayMask property 149
 - example 175
- exporting data 25
 - from a QueryDataSet 25
 - to text files 23
 - using patterns 175
- extractDataSet method 86
- extractDataSetChanges method 86
- extracting data 45

F

- fetchAsNeeded property 97
- fetching data 46
 - detail records 59, 97
 - from JDBC data sources 24, 61
 - optimizing 51
- fields
 - linking on common 94
 - required 154
- filtering
 - data 121, 124
- FilterRows sample 124
- flat file databases 23
- fonts 6
 - JBuilder documentation conventions 6
- formatted text files 24
 - importing 24
- formatter property
 - using 155
- formatting data 148
 - display masks for 149

G

- generating
 - database applications 117
- Group By clause 112
- Group By page
 - Data Modeler 112
- grouping data 144

H

- handling
 - errors 159
 - exceptions 159

I

- import masks 148, 149
- importing data 21
 - from text files 23
- indexes
 - database 165
 - unique vs. named 129
- insert procedures, custom 81
- insertProcedure property 81
- insertRow() method 67
- installing
 - InterClient 31
- InterBase
 - about 31
 - setting up for JBuilder 31
 - stored procedures example 63
 - stored procedures return parameters 83
 - tips 32
- Interbase 31
- InterBase and InterClient
 - using with JBuilder 32
- InterClient 31
 - about 31
 - connection errors 43
 - installing 31
 - setting up for JBuilder 31
 - using JDBC drivers 35
- INTERNALROW 85, 86
- Internet
 - developing client-server applications 31
- InternetBeans Express 17
- Intranet
 - developing client-server applications 31

J

- Java
 - database drivers 31
 - objects containing DataSets 85
 - RMI with databases 161
- Java data modules
 - saving queries 116
- Java Database Connectivity *See* JDBC
- Java interfaces
 - developing with InterClient 31
- JBCL components
 - data-aware 157
- JBuilder
 - newsgroups 8
 - reporting bugs 8

- JConnectionPool
 - optimizing performance 40
- JDataStore 19
 - creating 21
 - JDBC drivers 29, 31
 - operations 21
 - package 14
 - using 19
 - verifying 21
 - when to use 19
- JDataStore Explorer 20
- JDBC 1, 27, 78
 - pooling connections 38
- JDBC API 11
- JDBC connections 27
 - managing 14
 - manipulating traffic 172
 - monitoring 172
 - overview 27
 - tutorial 185
- JDBC data sources 24, 45, 61
 - accessing 27, 45
 - from text files 26
 - saving text file data 26
- JDBC drivers 28
 - adding to JBuilder 33
 - adding to project 34
 - InterClient 35
 - JDataStore JDBC drivers 29, 31
 - setting up 31
 - specified in database 29
 - when to use 19
- JDBC escape sequences 62
- JDBC Monitor 172
 - in applications 174
 - starting 172
 - using 174
- JdbNavField component 130
 - example 130
- JdbNavToolBar component, saving data 78
- JdbStatusLabel component 158
 - example 158
- JdbTable components, sorting data in 127
- JFC components 157
- joining tables 93

L

- libraries
 - adding to project 106
 - creating 33
 - required 106
- Link Queries dialog box (Data Modeler) 115
- linked tables
 - considerations 84
 - types 84
- linking data sets 93
- load options 47
- Load Options field
 - in QueryDescriptor 47
- loading data 67
- local databases, accessing 166
- Local InterBase Server 32
- locale
 - property 175

- locale-specific resources
 - loading 49
- locate method 132
- LocateOptions class 133
- locating data 121, 130, 133
 - column order 134
 - interactive 130
 - locate options 133
 - programmatically 132
 - variants 134
- lookup columns 136
 - creating 136
 - example 138
- lookup lists 135

M

- manipulating JDBC traffic 172
- many-to-many data relationships 84
- many-to-one data relationships 84
- masks 135
 - for data formats 149
 - for editing 149
 - for importing/exporting 149
- master tables 94
 - editing 98
- masterDataSet property 99
- master-detail relationships 115
 - creating 93, 99
 - defining 94
 - example 94
 - queries 59
 - resolving 100
 - custom 91
- MasterDetail sample 94
- masterLink property 94
- MasterLinkDescriptor class
 - usage overview 94
- metadata 69
 - discovery 69
 - exploring 165
 - obtaining 66
 - persisting 52, 71
 - setting as dynamic 72
 - updating in persistent columns 74
 - viewing 72
- metaDataUpdate property
 - with multiple tables 85
- middle-tier server implementations 85
- models
 - accessing information about 158
- MonitorButton
 - adding to palette 174
 - properties 174
 - using 174
- monitoring
 - connections 172, 174
 - JDBC drivers 172
- multi-column locates
 - column order 134
- multi-table resolution 83
 - resolution order 85
- multi-tier applications
 - deploying 164

N

- named
 - indexes 129
 - parameters 57
- navigating
 - multiple data sets 157
 - synchronizing components 157
- newsgroups 8
 - Borland and JBuilder 8
 - public 8
 - Usenet 8
- nontransient data members 86
- numeric data 24
 - importing 24
 - patterns 150
- numeric fields
 - exporting 175
- numeric patterns 148
 - examples of 150

O

- objects
 - containing DataSets 85
 - Java 155
 - storing 155
- one-to-many data relationships 84
- one-to-many relationships 93
- one-to-one data relationships 84
- opening data sets 52
- optimizing data retrieval 51
- Oracle PL/SQL stored procedures example 64
- Order By clause, adding 114
- Order By page 114

P

- packages
 - database-related 14
- parameter markers 57
- parameterized queries 53, 113, 114
 - adding columns 57
 - binding values 58
 - example 53
 - for master-detail records 59
 - supplying new values 59
- ParameterRow component 16, 54, 57
- parameters
 - return 83
 - specifying 48
- Parameters tab (QueryDescriptor) 48
- parsing
 - data 148
 - strings 149
- PARTIAL option
 - multi-column locates 134
- password, prompting for 38
- Paste Column button 113
- Paste Parameter button 113
- patterns 135, 148
 - boolean data 152
 - date data 150
 - examples of 150
 - for data entry 149
 - for exporting data 175

- numeric data 150
- string data 151
- time data 150
- performance enhancing data set 51
- Persist all Metadata option 71
- persistent columns 154
 - adding 75
 - controlling metadata update with 74
 - deleting 74
 - overview 73
- persisting
 - data 19, 153
- pick lists 135, 136
 - example 136
 - removing 137
- Place SQL Text In Resource Bundle
 - in QueryDescriptor 47, 49
- populating SQL tables 172
- private data members 86
- procedure calls
 - server-specific 62
- ProcedureDataSet component 11, 15
 - about 61
 - overview 15
 - resolving data 80
 - resolving example 80
 - saving changes to 81
 - sorting in 127
- ProcedureResolver component 77
 - coding 81
 - deleteProcedure property 81
 - insertProcedure property 81
 - properties 81
 - saving changes with 81
 - updateProcedure property 81
 - using 80
- procedures
 - resolving 80
- projects, adding
 - database drivers 34
- prompting for user name and password 38
- provideData method 67
- ProviderHelp, initData method 67
- providers
 - creating custom 65
 - custom 161
 - of data 45
- providing data
 - for database examples 122
 - from JDBC data sources 61
 - with parameterized queries 53

Q

- queries 46
 - building 47
 - column properties of multiple tables 85
 - containing WHERE clause 84
 - creating 166
 - creating parameterized 53
 - creating with Data Modeler 109
 - editing directly 114
 - ensuring updatability 52
 - executing 167
 - Group By clause 112
 - master-detail 115

- multiple in Data Modeler 115
- on multiple tables 84
- optimizing 73
- overview 46
- parameterized 113, 114
- required components 46
- saving to data modules 116
- sort order 114
- SQL
 - Database Pilot 167
- testing 114
- viewing results 114
- Where clause 113
- query property
 - editor 47
 - parameters 48
 - understanding 47
- query property, editor
 - tutorial 185
- Query tab 47
- QueryDataSet component 11, 15, 54
 - example 53, 78
 - exporting to a file 25
 - overview 15, 46, 51
 - query property setting 47
 - resolving changes 78
 - saving changes 79
 - sorting in 127
- QueryDescriptor component
 - Parameters tab 48
 - Place SQL text in resource bundle option 49
 - Query page 47
 - setting properties visually 47
- QueryProvider component 85
- QueryResolver component 77
 - adding 87
 - customizing 87, 88
 - default 87
 - events
 - controlling 88
 - for ProcedureDataSets 80
 - intercepting events 88
 - sample 78
 - saving changes with 80
 - using 78
 - with stored procedures 80

R

- read-only data sets 52
- reconciling data 77
- relational databases 93
- remote
 - servers 27
- remote databases 28
 - accessing 166
 - connecting to 28
- removing
 - persistent columns 74
 - tables 172
- reporting bugs 8
- required data 154
- resolution order
 - specifying 85
- resolution process
 - controlling 88

- ResolutionManager class 87
- resolveOrder property 83, 85
- resolver events 87
- ResolverEvents sample application 88, 89
- ResolverListener interface 88
- ResolverResponse 88
- resolvers 77
 - custom 77, 87, 90, 161
 - default 87
- resolving
 - example 81
 - ProcedureResolver 81
- resolving data 77, 78, 87, 90
 - customizing events 88
 - customizing resolver logic 87
 - default 87
 - handling errors 88
 - master-detail relationships 100
 - multiple tables 83
 - QueryDataSets 79
 - stored procedures 80
- resource bundles 47, 49
- resources
 - locale-specific 49
- retrieving data 27, 45, 65, 122
 - from a data module 103
 - from data sources 87
 - through stored procedures 61
- return parameters 83
- RMI
 - streaming data 85
 - with databases 161
- RowFilterListener interface
 - example 124
- rowID property
 - using 85
- RowIterator class 71

S

- saveChanges() method
 - and rowID property 85
- saving changes 78, 80, 87
 - master-detail relationships 100
 - to QueryDataSets 79
- saving data 77
 - example 81
 - JdbNavToolBar component 78
 - multiple tables 83
 - ProcedureResolver 81
 - using QueryResolver 80
- SCHEMA files 24
 - and exportDisplayMasks 149
- schemaName property 83
- Selected Sort Order Direction options 114
- serializing objects 85
- setResolver 87
- shared cursors 157
- SimpleStoredProcedure sample 80
- sort order 129
 - SQL queries 114
 - unique 129
- sort property editor 128
- sorting data 121, 127
 - in JdbTable components 127
 - in tables 127

- programmatically 130
- sort order 129
 - with design tools 128
 - with master-detail relationships 94
- special characters 149
- specifying resolution order 85
- SQL Builder 47
- SQL connections 27
- SQL databases
 - connecting to 28
- SQL queries 46
 - adding parameters 53
 - editing directly 114
 - ensuring updatability 52
 - Group By clause 112
 - master-detail 115
 - multiple in Data Modeler 115
 - optimizing 73
 - overview 46
 - required components 46
 - resourceable 49
 - saving to data modules 116
 - sort order 114
 - testing 114
 - view results 114
 - Where clause 113
- SQL servers, connecting to *See* SQL connections
- SQL Statement field
 - in QueryDescriptor 47
- SQL statements 62
 - defining 47
 - discussion of 62
 - encapsulating 61
 - examples 47
 - executing 167
- SQL tables
 - creating 170
 - deleting 172
 - from text files 25
 - populating 172
 - saving changes to 78
 - saving text file data 25
 - updating 77
- SqlRes class 49
- SQLResolver component 77, 87
 - customizing 87
 - for ProcedureDataSets 80
 - using ProcedureResolver 81
 - using with multiple tables 83
- status information 158
- status labels
 - adding to applications 158
- StatusEvent listener 159
- StorageDataSet component 15
 - adding empty columns 75
 - controlling column order 75
 - overview 15
 - saving changes 77
 - usage overview 45
- StorageDataSet methods
 - insertRow() 67
 - startLoading() 67
- storageDataSet property 152
- stored procedures
 - creating 62
 - example 81
 - examples 63, 64, 65
 - InterBase 83
 - overview 61
 - ProcedureResolver 81
 - resolving 80
 - return parameters 83
- streamable data sets 85
 - using 85
- StreamableDataSets sample, running 162
- streaming data 85
- string conversions, with masks 149
- string data, patterns 151
- string patterns 148
 - examples of 150
- strings
 - parsing 149
- summarizing data 144, 185
- Sybase stored procedures, example 65
- synchronizing components 157
- synonyms, displaying data in 168

T

- table data
 - editing 172
 - viewing 172
- tableColumnName property 83
- TableDataSet component 15
 - exporting formatted data 175
 - overview 15
 - resolving 25
 - saving changes to 25
 - sorting in 127
 - usage overview 23
- tableName property 83
- tables
 - creating 170
 - deleting 172
 - editing data 168
 - exploring 165
 - linked 84
 - not updatable 85
 - populating 172
 - querying 122
 - saving changes to 78
 - viewing data 168
- TestFrame.java sample 65
- testing
 - queries 114
- text fields, exporting 175
- text files 23
 - exporting 23, 25
 - importing 21, 23
 - to JDBC sources 26
 - to SQL tables 25
- TextDataFile component
 - resolving 26
 - retrieving JDBC data for 24
 - usage overview 23
- time data, patterns 150
- time fields, exporting 175
- time patterns 148
- time patterns, examples of 150
- totals, calculating 147
- transactions 78
 - default processing 77

tutorials

- aggregated data with calculated columns 185
- basic database application 185
- calculated aggregations 185
- importing and exporting data from a text file 175
- JDBC connections 185
- two-tier applications
 - generating 117

U

- unique indexes 129
- update procedure 81
- updateProcedure property 81
- updating
 - data from data sources 87
 - data sources 77
 - SQL tables 78
- URLs
 - adding in Data Modeler 110
 - opening in Data Modeler 110
- Use Data Module wizard 108
- Usenet newsgroups 8
- user input
 - controlling 149
 - parsing 149
- user name
 - prompting for 38

V

- ValidationException 159
- Variant class
 - locating data 134
- variant data types 155
- Variant.OBJECT data types
 - in columns 155
- VariantFormatter class 148
- views
 - displaying data in 168
 - of data 15

W

- Where clause 113
- Where page
 - Data Modeler 113

X

- XML database components 17

