

Database Application Developer's Guide



VERSION 5

Borland®
JBuilder™

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

Refer to the file DEPLOY.TXT 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. The furnishing of this document does not give you any license to these patents.

COPYRIGHT © 1997, 2001 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. Other product names are trademarks or registered trademarks of their respective holders.

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

Printed in the U.S.A.

JBE0050WW21001 5E7R0401

0102030405-9 8 7 6 5 4 3 2 1

PDF

Contents

Chapter 1		
Introduction	1-1	
Documentation conventions	1-5	
Chapter 2		
Understanding JBuilder database applications	2-1	
Understanding JBuilder's DataExpress architecture	2-4	
The Borland database-related packages.	2-6	
Chapter 3		
Setting up JBuilder for database applications	3-1	
Connecting to databases	3-2	
JBuilder sample files	3-2	
Note for Unix users	3-3	
Setting up JDataStore	3-4	
Setting up InterBase and InterClient.	3-4	
About InterBase and InterClient	3-4	
Using InterBase and InterClient with JBuilder	3-5	
Tips on using sample InterBase tables	3-6	
Adding a JDBC driver to JBuilder	3-7	
Creating the .library and .config files	3-7	
Adding the JDBC driver to projects	3-8	
Deploying database applications.	3-8	
Troubleshooting database connections in the tutorials.	3-9	
Chapter 4		
Connecting to a database	4-1	
Tutorial: Connecting to a database using the JDataStore JDBC driver	4-2	
Adding a Database component to your application.	4-3	
Setting Database connection properties	4-4	
Tutorial: Connecting to a database using InterClient JDBC drivers	4-6	
Common connection error messages.	4-9	
Using the Database component in your application	4-9	
Prompting for user name and password	4-9	
Pooling JDBC connections	4-10	
Optimizing performance of JConnectionPool	4-12	
Logging output	4-12	
Example	4-13	
Chapter 5		
Retrieving data from a data source	5-1	
When to use JDataStore vs. JDBC drivers	5-2	
Overview of the DataExpress components.	5-2	
Tutorial: An introduction to JBuilder database applications.	5-4	
Creating the application structure.	5-5	
Adding DataExpress components to your application	5-7	
Setting properties to connect the components	5-8	
Creating a user interface	5-9	
Connecting the DataExpress component to a UI component.	5-12	
Compiling, running, and debugging your application	5-12	
Summary	5-13	
Querying a database	5-14	
Tutorial: Querying a database using the JBuilder UI	5-16	
Retrieving data by querying a database	5-16	
Creating the UI	5-18	
Setting properties in the query dialog.	5-19	
The Query page	5-20	
The Parameters page	5-21	
Place SQL text in resource bundle	5-22	
Querying a database: Hints & tips.	5-24	
Enhancing data set performance	5-24	
Persisting query metadata	5-25	
Opening and closing data sets	5-26	
Ensuring that a query is updateable.	5-26	
Using parameterized queries to obtain data from your database	5-27	
Tutorial: Parameterizing a query	5-27	
Creating the application.	5-28	
Adding a Parameter Row	5-28	
Adding a QueryDataSet.	5-29	
Add the UI components.	5-30	

Parameterized queries: Hints & tips	5-33
Using parameters.	5-33
Re-executing the parameterized query with new parameters.	5-35
Parameterized queries in master-detail relationships.	5-35

Chapter 6 Using stored procedures **6-1**

Tutorial: Retrieving data using stored procedures	6-3
Creating tables and procedures for the tutorial.	6-3
Adding the DataSet components	6-4
Adding visual components	6-5
Stored procedures: Hints & tips	6-6
Discussion of stored procedure escape sequences, SQL statements, and server-specific procedure calls	6-7
Creating tables and procedures for the tutorial manually.	6-8
Stored procedures: InterBase, Oracle, and Sybase specific information	6-10
Example: Using InterBase stored procedures.	6-10
Example: Using parameters with Oracle PL/SQL stored procedures	6-10
Using Sybase stored procedures	6-11
Browsing sample applications that use stored procedures	6-12
Writing a custom data provider	6-12
Obtaining metadata	6-13
Invoking initData	6-14
Obtaining actual data	6-14
Tips on designing a custom data provider	6-15
Understanding the provideData method in master-detail data sets.	6-15

Chapter 7 Working with columns **7-1**

Understanding Column properties and metadata	7-1
Non-metadata Column properties	7-2
Viewing column information in the Column designer	7-2
The Generate RowIterator Class button	7-3

Using the Column designer to persist metadata	7-4
Making metadata dynamic using the Column designer	7-4
Viewing column information in the Database Pilot	7-5
Optimizing a query	7-6
Setting column properties	7-6
Persistent columns	7-7
Combining live metadata with persistent columns.	7-8
Removing persistent columns	7-8
Using persistent columns to add empty columns to a DataSet	7-9
Controlling column order in a DataSet	7-10

Chapter 8 Saving changes back to your data source **8-1**

Saving changes from a QueryDataSet	8-2
Tutorial: Adding a button to save changes from a QueryDataSet	8-3
Saving changes back to your data source with a stored procedure	8-5
Tutorial: Saving changes using a QueryResolver	8-6
Coding stored procedures to handle data resolution	8-8
Tutorial: Saving changes with a ProcedureResolver.	8-9
Example: Using InterBase stored procedures with return parameters	8-11
Resolving data from multiple tables	8-11
Considerations for the type of linkage between tables in the query	8-12
Table and column references (aliases) in a query string	8-13
Controlling the setting of the column properties.	8-13
What if a table is not updateable?	8-13
How can the user specify that a table should never be updated?	8-14
Using DataSets with RMI (streamable data sets)	8-14
Example: Using streamable data sets	8-14
Using streamable DataSet methods	8-15

Customizing the default resolver logic	8-16
Understanding default resolving	8-17
Adding a QueryResolver component	8-17
Intercepting resolver events	8-17
Using resolver events.	8-19
Writing a custom data resolver	8-20
Handling resolver errors	8-21
Resolving master-detail relationships	8-21

Chapter 9

Establishing a master-detail relationship 9-1

Defining a master-detail relationship	9-2
Fetching details.	9-3
Fetching all details at once	9-3
Fetching selected detail records on demand	9-3
Editing data in master-detail data sets.	9-4
Steps to creating a master-detail relationship.	9-5
Tutorial: Creating a master-detail relationship	9-6
Saving changes in a master-detail relationship	9-10
Resolving master-detail data sets to a JDBC data source	9-11

Chapter 10

Importing and exporting data from a text file 10-1

Tutorial: Importing data from a text file.	10-2
Adding columns to a TableDataSet in the editor	10-4
Importing formatted data from a text file.	10-4
Retrieving data from a JDBC data source	10-5
Exporting data	10-5
Tutorial: Exporting data to a text file.	10-6
Tutorial: Using patterns for exporting numeric, date/time, and text fields.	10-8
Exporting data from a QueryDataSet to a text file	10-10
Saving changes from a TableDataSet to a SQL table	10-10
Saving changes loaded from a TextDataFile to a JDBC data source.	10-11

Chapter 11

Using data modules to simplify data access 11-1

Creating a data module using the design tools	11-2
Create the data module with the wizard.	11-2
Add data components to the data module	11-3
Adding business logic to the data module	11-4
Using a data module	11-5
Adding a required library to a project.	11-5
Referencing a data module in your application	11-6
Understanding the Use Data Module dialog	11-7
Creating data modules using the Data Modeler	11-8
Creating queries with the Data Modeler	11-8
Opening a URL.	11-9
Beginning a query	11-10
Adding a Group By clause	11-12
Selecting rows with unique column values	11-13
Adding a Where clause	11-13
Adding an Order By clause.	11-14
Editing the query directly.	11-15
Testing your query.	11-15
Building multiple queries.	11-16
Specifying a master-detail relationship	11-16
Saving your queries	11-17
Generating database applications	11-18
Using a generated data module in your code	11-19

Chapter 12

Persisting and storing data in a DataStore 12-1

When to use a DataStore	12-1
Using the DataStore Explorer	12-2
DataStore operations	12-3

Chapter 13

Filtering, sorting, and locating data

13-1

Retrieving data for the tutorials	13-2
Filtering data	13-5
Tutorial: Adding and removing filters	13-6
Sorting data.	13-9
Sorting data in a JdbTable	13-9
Sorting data using the JBuilder visual design tools	13-10
Understanding sorting and indexing.	13-12
Sorting data in code	13-13
Locating data.	13-14
Locating data with a JdbNavField	13-14
Locating data programmatically	13-16
Locating data using a DataRow.	13-17
Working with locate options	13-17
Locates that handle any data type	13-19
Column order in the DataRow and DataSet.	13-19

Chapter 14

Adding functionality to database applications

14-1

Creating lookups.	14-2
Tutorial: Data entry with a picklist	14-3
Removing a picklist field	14-4
Tutorial: Creating a lookup using a calculated column	14-5
Using calculated columns.	14-7
Tutorial: Creating a calculated column in the designer	14-8
Aggregating data with calculated fields.	14-10
Tutorial: Aggregating data with calculated fields.	14-11
Setting properties in the AggDescriptor.	14-14
Creating a custom aggregation event handler.	14-15
Adding an Edit or Display Pattern for data formatting.	14-15
Display masks	14-17
Edit masks.	14-17
Using masks for importing and exporting data.	14-17

Data type dependent patterns	14-18
Patterns for numeric data	14-18
Patterns for date and time data.	14-19
Patterns for string data	14-20
Patterns for boolean data	14-21
Presenting an alternate view of the data	14-22
Ensuring data persistence	14-23
Making columns persistent.	14-24
Using variant data types	14-26
Storing Java objects	14-26

Chapter 15

Creating a user interface using dbSwing components

15-1

Tutorial: Using dbSwing components to create a database application UI.	15-2
Blocking editing in JdbTable.	15-5

Chapter 16

Using other controls and events

16-1

Synchronizing visual components	16-1
Accessing data and model information from a UI component	16-2
Displaying status information.	16-2
Building an application with a JdbStatusLabel component	16-3
Running the JdbStatusLabel application	16-4
Handling errors and exceptions.	16-4
Overriding default DataSetException handling on controls	16-5

Chapter 17

Creating a distributed database application using DataSetData

17-1

Understanding the sample distributed database application (using Java RMI and DataSetData).	17-2
Setting up the sample application	17-3
What is going on?	17-3
Passing metadata by DataSetData	17-4
Deploying the application on 3-tiers.	17-4
For more information	17-5

Chapter 18	
Database administration tasks	18-1
Exploring database tables and metadata	
using the Database Pilot	18-1
Browsing database schema objects	18-2
Setting up drivers to access remote	
and local databases	18-3
Executing SQL statements	18-4
Using the Explorer to view and edit	
table data	18-5
Using the Database Pilot for database	
administration tasks	18-7
Creating the SQL data source	18-7
Populating a SQL table with data using	
JBuilder	18-9
Deleting tables in JBuilder	18-9
Monitoring database connections	18-9
Understanding the JDBC Monitor user	
interface	18-10

Using the JDBC Monitor in a running	
application	18-10
Adding the MonitorButton to the	
Palette	18-11
Using the MonitorButton Class from	
code	18-11
Understanding MonitorButton	
properties	18-11

Chapter 19	
Sample database application	19-1
Sample international database application. . .	19-2

Appendix A	
Database FAQ	A-1

Index	I-1
--------------	------------

Tutorials

Tutorial: Connecting to a database using the JDataStore JDBC driver	4-2	Tutorial: Creating a master-detail relationship	9-6
Tutorial: Connecting to a database using InterClient.	4-6	Tutorial: Importing data from a text file	10-2
Tutorial: An introduction to JBuilder database applications.	5-4	Tutorial: Exporting data to a text file	10-6
Tutorial: Querying a database using the JBuilder UI	5-16	Tutorial: Using patterns for exporting numeric, date/time, and text fields	10-8
Tutorial: Parameterizing a query	5-27	Tutorial: Adding and removing filters	13-6
Tutorial: Retrieving data using stored procedures	6-3	Tutorial: Data entry with a picklist	14-3
Tutorial: Adding a button to save changes from a QueryDataSet	8-3	Tutorial: Creating a lookup using a calculated column.	14-5
Tutorial: Saving changes using a QueryResolver	8-6	Tutorial: Creating a calculated column in the designer	14-8
Tutorial: Saving changes with a ProcedureResolver	8-9	Tutorial: Aggregating data with calculated fields	14-11
		Tutorial: Using dbSwing components to create a database application UI.	15-2

Introduction

Database application development is a feature of JBuilder Professional and Enterprise.

Distributed application development is a feature of JBuilder Enterprise.

The *Database Application Developer's Guide* provides information on using JBuilder's DataExpress database functionality to develop database applications. It also explains using 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://forums.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://forums.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 the java.sun JDBC API. Some versions of JBuilder provide additional DataExpress components (on the DataExpress tab of the component palette) that greatly simplify development of database applications, as described in this book.

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. The JDataStore component provides a replacement for MemoryStore that 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 definitions of any unfamiliar Java terms, see “Java glossaries” in Part I, “Quick Start” of *Learning Java with JBuilder*.

To create a database application in JBuilder, you need to:

- **Understand JBuilder’s DataExpress architecture.**

Chapter 2, “Understanding JBuilder database applications” introduces the DataExpress architecture, describes JBuilder’s set-oriented approach to handling data, and provides an overview of the main data components in the DataExpress package.

- **Set up JBuilder for database applications.**

Chapter 3, “Setting up JBuilder for database applications” provides the setup information required to step through and run the sample applications referenced in this manual. This includes JBuilder setup for access of sample data using JDataStore JDBC driver and JBuilder sample files.

- **Connect to a database.**

Chapter 4, “Connecting to a database” describes how to connect your database components to a server.

- **Retrieve data from a server.**

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.

For most applications, you will want to use a data module to hold the DataExpress package components. Chapter 11, “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.

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 10, “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.

- **Decide how to store your data.**

Chapter 12, “Persisting and storing data in a DataStore” discusses using JDataStore components for organizing an application’s StorageDataSet’s, files, and serialized JavaBeans/Objects states into a single, all Java, portable, compact, high-performance, persistent storage mechanism. For information beyond the scope of this book, see the *JDataStore Developer’s Guide*.

- **Save changes to your data.**

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*).

Chapter 10, “Importing and exporting data from a text file” explains how to provide data to your application from a text file, and to save the data back to a text file or to a SQL data source.

- **Manipulate your data.**

These chapters describe features that enhance database applications, and how to use the features.

- 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 13, “Filtering, sorting, and locating data” explains the differences between these features, and provides a tutorial for each as well.
- Chapter 14, “Adding functionality to database applications” includes
 - 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 15, “Creating a user interface using dbSwing components” shows how to use dbSwing components to create a user interface for your application.
- Chapter 18, “Database administration tasks” includes such common database tasks as
 - 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
 - handling errors and exceptions in your application

To aid in your understanding of database applications, you may also wish to:

- **View a sample database application.**

Chapter 19, “Sample database application” consists of a complete sample database application that ties in individual features described in greater detail in the previous chapters. Run this application to see various DataExpress package database features in action.

Chapter 17, “Creating a distributed database application using DataSetData” discusses using DataExpress components in a distributed object computing environment (using Java RMI).

For deploying database applications, you may wish to consider using:

- **Servlets**

Servlets are server-side versions of applets, or a server-side Java program that is initiated when certain HTML is encountered. This section describes how to create a servlet in JBuilder, provides a tutorial for practice, and provides links to sample servlets on other web sites. See “Working with Servlets” in *Web Application Developer’s Guide* for more information.

- **JavaServer Pages (JSP).**

JavaServer Pages (JSP) technology provides an easy and powerful way to build web pages with dynamically-generated content. JSP technology is created using HTML-like tags and scriptlets written in Java. Standard HTML or XML commands handle formatting and design. This section describes how to create a JavaServer Page in JBuilder, including a tutorial for practice. See “Developing JavaServer Pages” in *Web Application Developer’s Guide* for more information.

A document on the Borland web site contains questions and answers culled from the JBuilder database newsgroup. See Appendix A, “Database FAQ” for information on accessing both this document and the newsgroup.

Documentation conventions

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

Table 1.1 Typeface and symbol conventions

Typeface	Meaning
Monospace type	<p>Monospaced type represents the following:</p> <ul style="list-style-type: none"> • text as it appears onscreen • anything you must type, such as “Enter Hello World in the Title field of the Application wizard.” • file names • path names • directory and folder names • commands, such as SET PATH, CLASSPATH • Java code • Java identifiers, such as names of variables, classes, interfaces, components, properties, methods, and events • package names • argument names • field names • Java keywords, such as void and static
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 and for book titles.
<i>Keycaps</i>	This typeface indicates a key on your keyboard. For example, “Press <i>Esc</i> to exit a menu.”
[]	Square brackets in text or syntax listings enclose optional items. Do not type the brackets.
< >	Angle brackets in text or syntax listings indicate a variable string; type in a string appropriate for your code. Do not type the angle brackets. Angle brackets are also used for HTML tags.
...	An ellipsis in syntax listing indicates code that is missing from the example.

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

Table 1.2 Platform conventions and directories

Item	Meaning
Paths	All paths in the documentation are indicated with a forward slash (/). For the Windows platform, use a backslash (\).
Home directory	The location of the home directory varies by platform. <ul style="list-style-type: none"> • For UNIX and Linux, the home directory can vary. For example, it could be <code>/user/[username]</code> or <code>/home/[username]</code> • For Windows 95/98, the home directory is <code>C:\Windows</code> • For Windows NT, the home directory is <code>C:\Winnt\Profiles\[username]</code>
<code>.jbuilder</code> directory	The <code>.jbuilder</code> directory, where JBuilder settings are stored, is located in the home directory.
<code>jbproject</code> directory	The <code>jbproject</code> directory, which contains project, class, and source files, is located in the home directory. JBuilder saves files to this default path.
Screen shots	Screen shots reflect JBuilder's Metal Look & Feel on various platforms.

Understanding JBuilder database applications

Database application development is a feature of JBuilder Professional and Enterprise.

Distributed application development is a feature of JBuilder Enterprise.

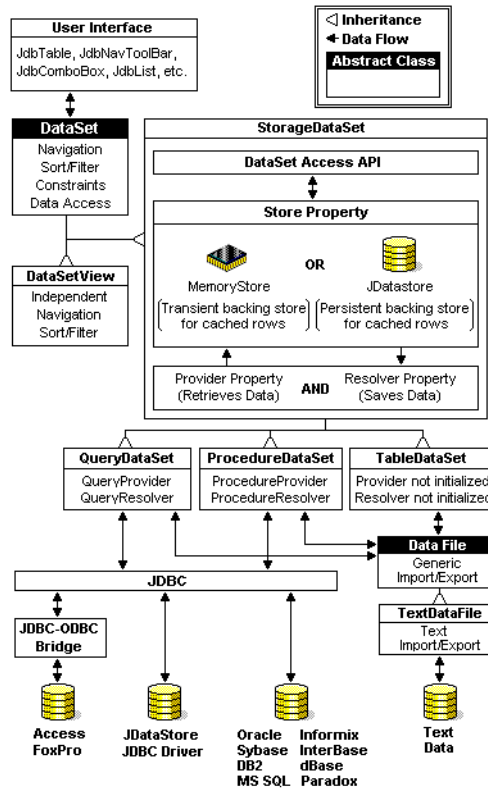
A database application is any application that accesses stored data and allows you to view and perhaps modify or manipulate that data. In most cases, the data is stored in a database. However, data can also be stored in files as text, or in some other format. JBuilder allows you to access this information 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.

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:



The major components in a database application are:

- DataSet**

DataSet is an abstract class. A large amount of the public API for all DataSets is surfaced in this class. All navigation, data access, and update APIs for a DataSet are surfaced in this class. Support for master-detail relationships, row ordering, and row filtering are surfaced in this class. Some of the dbSwing data-aware components have a dataSet property. This means that a JdbTable, for example, can have its dataSet property set to the various extensions of DataSet: DataSetView, QueryDataSet, ProcedureDataSet, and TableDataSet.
- StorageDataSet**

StorageDataSet can use in-memory storage (MemoryStore) to cache its data. The StorageDataSet store property can also be set to a DataStore component to provide persistence for the DataSet data. The StorageDataSet manages the storage of DataSet data, indexes used to

maintain varying views of the data, and persistent Column state. All structural APIs (add/delete/change/move column) are surfaced in this class. Since StorageDataSets manage the data, it is where all row updates, inserts, and deletes are automatically recorded. Since all changes to the StorageDataSet are tracked, we know exactly what needs to be done to save (resolve) these changes back to the data source during a resolution operation.

- DataStore

The DataStore component provides high performance data caching and compact persistence for DataExpress DataSets, arbitrary files, and Java Objects. The DataStore component uses a single file to store one or more data streams. A DataStore file has a hierarchical directory structure that associates a name and directory status with a particular data stream.

- DataSetView

This component can be used to provide independent navigation (a cursor) with row ordering and filtering different than that used by the base DataSet. To use this component, set the storageDataSet property of the DataSetView component. Use this component when multiple components need to dynamically switch to a new DataSet. The components can all be wired to the same DataSetView. To force them all to view a new DataSet, the DataSetView storageDataSet property can be changed.

- QueryDataSet

This is a JDBC specific DataSet. It manages a JDBC provider of data. The data to be provided is specified in a query property. The query property specifies a SQL statement.

- ProcedureDataSet

This is a JDBC specific DataSet. It manages a JDBC provider of data. The data to be provided is provided with a procedure property. The procedure property specifies a stored procedure.

- TableDataSet

This is a generic DataSet component without a built-in provider mechanism. Even though it has no default provider, it can be used to resolve its changes back to a data source. Column and data can be added to a TableDataSet through DataSet methods or by importing data with a DataFile component (like TextDataFile).

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 class hierarchy associated with the `DataSet` methods is as follows:

```
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 columns in the current row
- `ReadRow` methods give read access to columns in the current row
- `TableDataSet` and `QueryDataSet` inherit all these methods

The next section, entitled “Understanding JBuilder’s DataExpress architecture,” discusses the components of the `DataExpress` architecture in more detail.

Understanding JBuilder’s DataExpress architecture

`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 DataStore.

MemoryStore caches all data and data edits in memory. DataStore uses an all Java, small footprint, high performance, embeddable database to cache data and data edits. The DataStore 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 an 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 DataSets 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 DataSets, 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 `DataStore` 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.

The Borland database-related packages

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	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. See Chapter 4, “Connecting to a database” for more description and a tutorial using this component.	The <code>ConnectionDescriptor</code> object stores connection properties of driver, URL, user name, and password. Accessed using the <code>connection</code> property.
DataSet	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.	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 13-9 for a tutorial. 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 a tutorial.

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 <code>InterBase</code> or <code>Oracle</code>), 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>DataStore</code> provides high performance data caching and compact persistence for <code>DataExpress</code> <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>DataStore</code> file has a hierarchical directory structure that associates a name and directory status with a particular data stream. <code>DataStore</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 Chapter 12, "Persisting and storing data in a <code>DataStore</code>," and the <i>JDataStore Developer's Guide</i>, for more description of the <code>DataStore</code> component.</p>	<p>Caching and persisting <code>StorageDataSets</code> 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>DataStore</code>. The driver supports both local and remote access. Both types of access require a user name (any string, with no setup required) and an empty password.</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 5-14 for more description and a tutorial 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 description and a tutorial using 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>

Component/Class	Description	Frequently used properties
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 10, “Importing and exporting data from a text file” for more description and a tutorial using 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 14-22 for more description and a tutorial using 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>
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 a tutorial using 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 <code>Column</code> designer (properties only) or in the <code>Inspector</code> 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>	

Component/Class	Description	Frequently used properties
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 5-27 for more description and a tutorial 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 dataset components grouped into a data model. You create a new, empty data module by selecting the Data Module icon from the File New dialog. 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 11, “Using data modules to simplify data access” for more description and a tutorial 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.

Setting up JBuilder for database applications

Database application development is a feature of JBuilder Professional and Enterprise.

Distributed application development is a feature of JBuilder Enterprise.

If you downloaded JBuilder, you also need to download the Samples Pack in order to have the samples. Note that some samples only work with JBuilder Enterprise.

To operate the database tutorials included in this book, you'll need to install the `JDataStore` JDBC driver and/or the `InterClient` JDBC driver. This topic discusses setting up `JDataStore` and `InterClient` for use in the tutorials. See "Adding a JDBC driver to JBuilder" on page 3-7.

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 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 (<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.

JBuilder sample files

The JBuilder/samples directory contains files for various tutorials and examples presented in this manual. The DataExpress sample files are located in the /samples/DataExpress/ subdirectory of your JBuilder installation.

Note If you downloaded JBuilder, you also need to download the Samples Pack in order to have the samples. Note that some samples only work with JBuilder Enterprise.

The database sample applications contain samples that access data from a DataStore, `employee.jds`, and from the InterBase sample database `employee.gdb`. For more information on DataStore, see *JDataStore*

Developer's Guide. For more information on InterBase Server, refer to its on-line documentation.

If you wish to examine the sample applications in the JBuilder designer, please note that you should build the project for each sample before bringing it into the designer. To do this, select Project | Rebuild Project.

Note for Unix users

If you install JBuilder as 'root', but run JBuilder under your normal user account, you will not be able to run any of the samples directly from the JBuilder samples directory. Attempting to do so will result in a compile-time error, Error #: 914: unable to write to output directory.

In order to run a sample project, you must have read/write access to all files and directories created or used by the sample. In general, if you only have read-only access to the JBuilder samples subdirectory, you will need to copy the samples into a directory in which you have read/write access to run the samples. Because some samples require files (e.g., databases) in other subdirectories of the samples directory, it is recommended that you make your own copy of the entire samples tree, if possible. Samples which try to access files in other samples directories write a message indicating the directory in which they expect to find a file.

In order to run the samples in such an environment, copy the entire samples tree to another directory to which you have full read/write access. While it is possible to copy projects individually from the sample tree, there are several samples which require access to files in peer-level directories. Such samples have been modified to automatically look for such files relative to the default sample tree hierarchy, and to display a message at run time indicating the path being used to locate such files.

An easy way to copy the entire samples tree is to use the `cp -R` command. For example, to copy the tree to a 'samples' subdirectory of your home directory, do the following:

```
% cd
% cp -R /usr/local/jbuilder/samples .
```

Alternatively, you could run the shell script in the main samples directory named 'chmod_samples'. Running this script enables the user who installs JBuilder to control access to the samples.

```
Usage: chmod_samples full
        to allow all users to compile, run, and modify sample files
or chmod_samples run
        to allow all users to compile, run, but not modify sample files
or chmod_samples read
        to allow all users to read but not run or modify sample files
```

Setting up JDataStore

DataStore Explorer is disabled on the Tools menu when you first install JBuilder Foundation or Professional. To enable DataStore Explorer and DataStore Server under the Tools menu, and to include the DataStore components on the Data Express tab of the component palette, you need to do one or more of the following:

- Install JDataStore on top of JBuilder Professional into the same location as JBuilder. (JBuilder Enterprise installs JDataStore automatically, making this step unnecessary.)
- Enter a valid JDataStore Developer license. Choose File | License Manager within DataStore Explorer to enter the JDataStore license.

Installing JDataStore installs the JDataStore Server on your local machine. No special configuration is necessary. The DataStore library will be added to your project when you connect to a JDataStore database. The first time you connect using JDataStore, you will be prompted for your serial number and password.

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

For more information on the DataStore Explorer, see the *JDataStore Developer's Guide*.

Setting up InterBase and InterClient

About 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.

InterClient includes a server-side driver, called *InterServer*. It can be downloaded from www.interbase.com. This server-side middle ware serves as a translator between the InterClient-based clients and the

InterBase database server. It includes the Java application development classes, and a web server deployment kit.

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 their instructions, then start the InterBase Server, followed by InterClient's InterServer.

If you have trouble connecting, be sure the InterBase database and InterServer are both running. InterServer and the 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 and InterServer are on a different platform than JBuilder, you need to:

- Make sure InterBase and InterServer are 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 and InterServer.

After InterClient is installed, add it to JBuilder using Tools | Enterprise Setup, then add it to your required list of libraries for your project in Project | Properties. For more information, see “Adding a JDBC driver to JBuilder” on page 3-7.

Tips on using sample InterBase tables

- Sample databases are installed by the setup program. You may wish to make a copy of the employee.jds sample database so that you can easily restore the file to its original condition after experimenting with database programming.
- These sample databases enforce many constraints on data values, as is normal in a realistic application.
 - The EMPLOYEE table is used extensively in the examples in this manual. Constraints on the employee table include:
 - 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.1
 - 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.

Basically, it's safest to modify the LAST_NAME, FIRST_NAME, PHONE_EXT fields in existing records.

- The CUSTOMER table is also used in the database tutorials. Its constraints include:
 - CUST_NO is generated, so no need to input for new records.

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, for example, see Chapter 8, “Saving changes back to your data source.”

To view the metadata for the sample tables,

- Run Tools | Database Pilot.

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 Tools | Enterprise Setup. Click the Database Drivers tab which displays .config files for all the currently known database drivers.
- 2 Click Add to add a new driver, then New to first 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.

- 3 Type a name and select a location for the new file in the Create New Library dialog box.
- 4 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.
- 5 Click OK to close the file browser. This displays the new library at the bottom of the library list and selects it.
- 6 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).

- 7 Select the new .config file in the database driver list and click OK. This places the .config file in the JBuilder /lib/ext directory.
- 8 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, then click the Add button.
- 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 | 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 “Tutorial: Connecting to a database using InterClient JDBC drivers” on page 4-6.

Deploying database applications

How can I package my JDBC driver with my application or applet? See the online help topic “Deploying programs and applets: Deploying JDBC drivers” for the most up-to-date information on deploying database applications.

Troubleshooting database connections in the tutorials

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 “Debugging DataStore applications” in the *JDataStore Developer’s Guide*.
- Check the Borland JDataStore FAQ at <http://www.borland.com/jdatastore/productinfo/faq.html>.
- Read “Common connection error messages” in Chapter 4, “Connecting to a database” for problems connecting to InterBase via InterClient.

Connecting to a database

Database application development is a feature of JBuilder Professional and Enterprise. Distributed application development is a feature of JBuilder Enterprise.

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 (any string, with no setup required).

For information on connecting to a database using the DataStore driver, see "Tutorial: Connecting to a database using the JDataStore JDBC driver" on page 4-2.

- 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 "Tutorial: Connecting to a database using InterClient JDBC drivers" on page 4-6.

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.

Tutorial: Connecting to a database using the JDataStore JDBC driver

This tutorial assumes you are familiar with the JBuilder design tools. If not, see the online help topic "Designing a user interface."

This tutorial outlines:

- "Adding a Database component to your application" on page 4-3
- "Setting Database connection properties" on page 4-4
- "Using the Database component in your application" on page 4-9
- "Prompting for user name and password" on page 4-9
- "Pooling JDBC connections" on page 4-10

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 11, “Using data modules to simplify data access.”

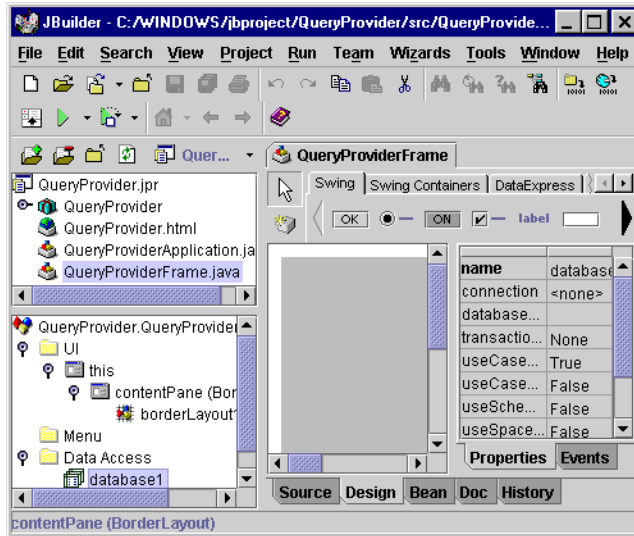
To add the Database component to your application,

- 1 Create a new project and application files using the Application wizard. (You can optionally follow this tutorial to add data connectivity to an existing project and application.) To create a new project and application files:
 - 1 Select `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.
 - 2 Select `File | New` and double-click the Application icon. Accept (or modify) all defaults.
- 2 Open the UI designer by selecting the file `Frame1.java` in the content pane, then select the Design tab at the bottom of the AppBrowser.
- 3 Select the DataExpress tab from the component palette. 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 content pane, looking like this:



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.

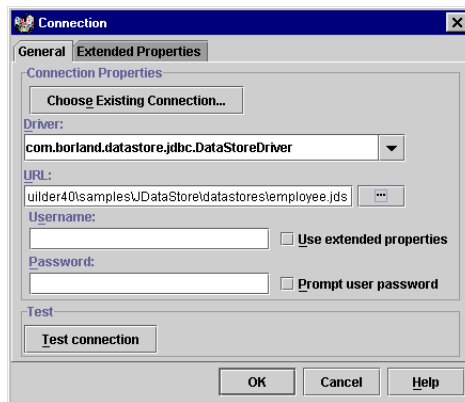
The following steps describe how to set connection properties through the UI designer to the sample `JDataStore` database.

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 3-4.

- 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	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 DataStore 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: Unix: <code>jdbc:borland:dslocal:/usr/local/jbuilder/samples/JDataStore/datastores/employee.jds</code> Windows: <code>jdbc:borland:dslocal:C:\jbuilder\samples\JDataStore\datastores\employee.jds</code>
Username	The user name authorized to access the server database. For the sample tutorials, any user name will work.
Password	The password for the authorized user. For the tutorials, no password is required.
Prompt user for password	Whether to prompt the user for a password when opening the database connection.

The dialog looks like this:



- 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 DataStore was installed with JBuilder, you will be prompted for your serial number and password the first time you use it. If you don't have the DataStore information at this time, you can enter it later by selecting Tools | DataStore Explorer, then selecting License Manager from its File menu.

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 3-4 for more information.

- 5 Click OK to exit the dialog 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:

```
database1.setConnection(new
    com.borland.dx.sql.dataset.ConnectionDescriptor(
        "jdbc:borland:dslocal:
        /usr/local/jbuilder/samples/JDataStore/datastores/employee.jds",
        , , false, "com.borland.datastore.jdbc.DataStoreDriver");
```

- 6 Select a DBDisposeMonitor component from the More dbSwing tab. Click in the content pane to add it to the application. The DBDisposeMonitor will close the DataStore when the window is closed.
- 7 Set the DBDisposeMonitor's dataAwareComponentContainer property 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 DataStore, and to execute SQL statements, and browse and edit data in existing tables.

Tutorial: 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 11, “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 3-4 and “Adding a JDBC driver to JBuilder” on page 3-7 to make sure your system is correctly set up for accessing the sample InterBase files.
- 2 Make sure InterServer is running (it should be).
- 3 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 selecting File | New, and double-clicking the Application icon. Select all defaults.

JBuilder will create the necessary files and display them in the AppBrowser project pane. The file `Frame1.java` will be open in the content pane. `Frame1.java` will contain the user interface components for this application.

- 4 Click the Design tab on `Frame1.java` at the bottom of content pane.
- 5 Select the Data Express tab on the component palette, and click the `Database` component.
- 6 Click anywhere in the content pane or UI designer to add the `Database` component to your application.
- 7 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.

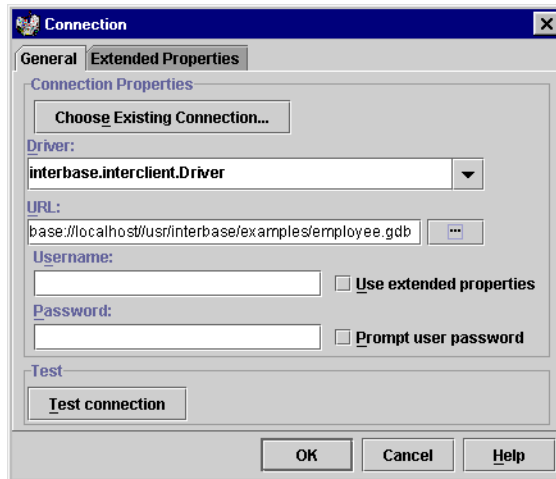
To set the `connection` property,

- 1 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.

2 Set the following properties:

Property	Value
Driver	interbase.interclient.Driver
URL	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: Unix: <code>jdbc:interbase://localhost//usr/interbase/examples/employee.gdb</code> Windows: <code>jdbc:interbase://localhost/D:\InterBaseCorp\InterBase\examples\database\employee.gdb</code>
Username	SYSDBA
Password	masterkey

The dialog looks like this:



- 3 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 4-9 for solutions to some typical connection problems.
- 4 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.

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. Select 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.

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 tutorials on how to use these components, see:

- "Querying a database" on page 5-14
- "Using parameterized queries to obtain data from your database" on page 5-27
- Chapter 6, "Using stored procedures"

Most of the sample applications and tutorials use a Database connection to the sample EMPLOYEE DataStore, as described here. A few of the sample applications and tutorials, particularly those that use stored procedures to retrieve or save data, use a connection to the InterBase employee database through the InterClient JDBC driver.

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 11, "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 or the `javax.sql` standard extensions package. If your version of `JBuilder` does not include `J2EE.jar` or `jdbc2_0stdext.jar`, you will need to download one of them from Sun, and add it to your project as a required library. See "Adding a required library to a project" on page 11-5 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 `DataSource`, 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 Inprise Application 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`.

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 hex values displayed in [] in the log files are `hashCode()` values for an `Object`.

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 11, “Using data modules to simplify data access.”

```
import com.borland.dx.dataset.*;
import com.borland.dx.sql.dataset.*;
import com.borland.javax.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;
    private static Database databasel;

    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 DataStore file. The specific
        // URL will depend on the location
        // of your DataStore 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);
    }
}
```

Pooling JDBC connections

```
// Instantiate a Database object
database1 = new Database();
// Assign the pool as the data source
// for the Database object
database1.setDataSource(pool);
}

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

public static JdbcConnectionPool getPool() {
    return pool;
}

public static Database getDatabase() {
    return database1;
}
}
```

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();
    }
}
}
}
```


Retrieving data from a data source

Database application development is a feature of JBuilder Professional and Enterprise.

Distributed application development is a feature of JBuilder Enterprise.

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 tutorials 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.

When to use JDataStore vs. JDBC drivers

You may wish 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

You may wish 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

Overview of the DataExpress components

This chapter discusses retrieving data using the DataExpress components listed below. There are tutorials showing the steps necessary to use these components to create database applications. The samples are located in `JBuilder/samples/DataExpress/`. If you experience problems running the sample applications, see “JBuilder sample files” on page 3-2 for information critical to this process.

- `TextDataFile`

The `TextDataFile` component specifies the properties of a text file that affect its import and export, such as delimiters, field separators, and so on. This component is used when:

- importing data stored in a text format into a `TableDataSet` component
- exporting the data stored in any `StorageDataSet` to a text file

“Tutorial: An introduction to JBuilder database applications” on page 5-4 steps you through creating a database application and user interface, even if you are not connected to any SQL or desktop databases. This tutorial uses data in a text file that ships with JBuilder.

For a tutorial that steps you through creating a database application using your own text file, see Chapter 10, “Importing and exporting data from a text file.”

- QueryDataSet

The `QueryDataSet` component provides functionality to run a query statement (with or without parameters) against a table in a SQL database.

“Querying a database” on page 5-14 steps through creating a local copy of data by executing a query and storing its results in a `QueryDataSet` component.

“Using parameterized queries to obtain data from your database” on page 5-27 outlines the steps required when adding parameters to your query statement.

- ProcedureDataSet

The `ProcedureDataSet` component provides functionality to run a stored procedure against data stored in a SQL database, passing in parameters if the procedure expects them. The procedure call is expected to return a cursor or output that can be used to generate a `DataSet`.

Chapter 6, “Using stored procedures” steps through creating a local copy of data by executing a stored procedure and storing its result set in a `ProcedureDataSet` component.

- TableDataSet

The `TableDataSet` component may or may not have a formal provider or resolver of its data. Its properties allow it to import file-based data. Use this component to create a `StorageDataSet` from sources other than SQL databases, for example, by importing data stored in a text file, from data computations, or to simply work with database data off-line. You can also use this component, or any other component that extends from `StorageDataSet`, to directly access tables stored in a `DataStore` database file.

Chapter 10, “Importing and exporting data from a text file” describes how to import data from a text file into the `TableDataSet` component. This topic discusses how to programmatically add `Column` components and read in data such as date and timestamp data for which there are no standards.

- Any DataSet

The `DataSet` class is an abstract class that provides basic editing, viewing, and cursor functionality for access to two-dimensional data.

“Writing a custom data provider” on page 6-12 discusses custom data providers, and how they can be used as providers for a `TableDataSet` and any `DataSet` derived from `TableDataSet`.

- Column

The `Column` component stores important properties such as data type and precision, as well as visual properties such as font and alignment. For `QueryDataSet` and `ProcedureDataSet` components, `Column` components are dynamically created each time the `StorageDataSet` is instantiated, mirroring the actual columns in the data source at that time.

Chapter 7, “Working with columns” discusses column properties, persistent columns, and metadata.

For `QueryDataSet` and `ProcedureDataSet` components, the data source is often a SQL server database. In this case, you also need a `Database` component to handle the connection to the server. See Chapter 4, “Connecting to a database” for more information on connecting to a server. When using the `TableDataSet` and `TextDataFile` components, you are usually reading data from a text file. Because you are not accessing SQL server data, you do not need a `Database` component.

See also

- “Understanding JBuilder’s DataExpress architecture” on page 2-4
- *DataExpress Component Library Reference*
- “Exploring database tables and metadata using the Database Pilot” on page 18-1
- Chapter 11, “Using data modules to simplify data access”

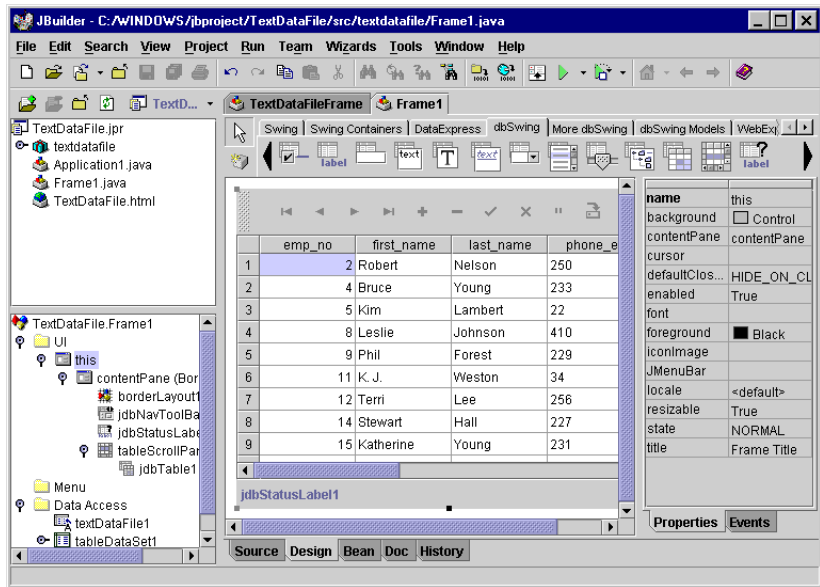
Tutorial: An introduction to JBuilder database applications

This introductory tutorial steps you through creating a basic database application and user interface (UI) using the JBuilder design tools to read data from a text file. You can create a database application even if you are not connected to a SQL or desktop database. You can create the database application using data in a text file (in this case, a data file that ships with JBuilder) and the `DataExpress TextDataFile` and `TableDataSet` components.

This tutorial then explores the JBuilder `DataExpress` architecture and applies database concepts to the tutorial even though this application does not connect to a SQL database. The UI for this application consists of three components:

- A table component that displays the data from the text file.
- A toolbar to aid in browsing through the data.
- A status area that displays messages.

When you have completed the tutorial, your application will look like this:



The completed application can be viewed by opening the sample project file, `TextDataFile.jpr`, in `/jbuilder/samples/DataExpress/TextDataFile/`.

Note If you downloaded JBuilder, you must also download the Samples Pack to get the completed sample.

For a tutorial that steps you through creating a database application using your own text file, see Chapter 10, “Importing and exporting data from a text file.”

Creating the application structure

The first step when creating an application in JBuilder is to set up the framework for the application. You can use JBuilder’s visual design tools to quickly create the application structure and, in future steps of this tutorial, to read data from the text file and to develop the user interface.

To create the application structure,

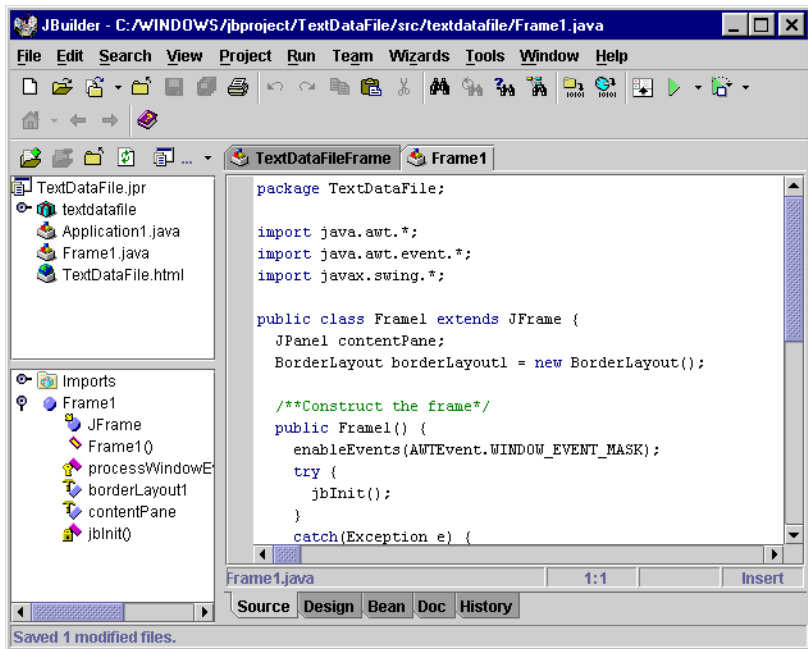
- 1 Choose `File | Close All` from the JBuilder menu to close any existing projects.
- 2 Choose `File | New Project` to start the Project wizard.
- 3 Modify the project name to be `TextDataFile.jpr`.

- 4 Click Next and, optionally, enter information for a title (TextDataFile tutorial), your name, your company, and a project description (This tutorial...).
- 5 Click Finish.

Now that the basic project structure is created, open the Application wizard to create a new Java application shell which contains a *Frame* that will be the UI container.

- 1 Select File | New from the menu. Double-click the Application icon. The first page of the Application wizard displays.
- 2 Accept the defaults.
- 3 Click Next.
- 4 Enter TextDataFile Tutorial in the Title field.
- 5 Click Finish.
- 6 Choose File | Save All.

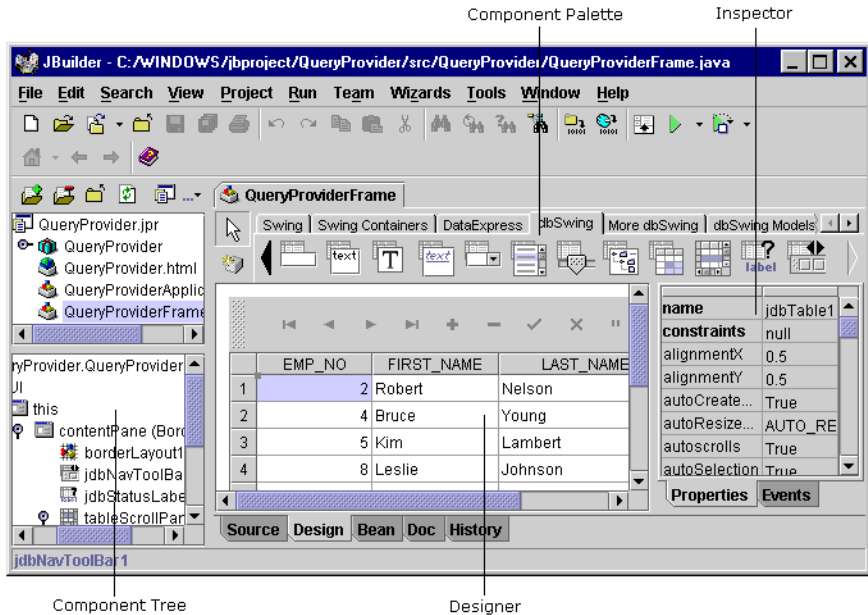
The `Application1.java` and `Frame1.java` files are added to the project and are displayed in the project pane (top left pane). `Frame1.java` is opened by default in the content pane with its Source tab selected, and its structure displayed in the structure pane (bottom left pane).



Adding DataExpress components to your application

You can use the JBuilder visual design tools to add `DataExpress` components to your application. The visual design tools are available on the Design tab in the content pane.

Figure 5.1 JBuilder visual design tools



You add `DataExpress` components by clicking them on the `DataExpress` tab of the component palette, then dropping them into either the UI designer or the component tree (lower left pane of the AppBrowser in design view). JBuilder will automatically generate the corresponding source code for you.

Even though the `DataExpress` components are not visual, JBuilder displays them in the `DataExpress` folder in the component tree so you can select them and set their properties using the Inspector.

For more information on the JBuilder visual design tools, see “Designing a User Interface” in *Building Applications with JBuilder*.

You will need two `DataExpress` components to import data from a text file:

- A `TextDataFile` to connect the `TableDataSet` to a text file.
- A `TableDataSet` to read a table (rows and columns) of data from a data source (in this case, the text file) and to work with the data in the JBuilder `DataSet` format.

Follow the steps below to add these components to your application:



- 1 Click the Design tab at the bottom of the content pane for `Frame1.java`.
- 2 Click the `TextDataFile` component on the DataExpress tab of the component palette.
- 3 Click anywhere in the component tree to add it to your application. `textDataFile1` appears in the DataExpress folder of the component tree.



- 4 Click the `TableDataSet` component and add it to the application. It appears as `tableDataSet1`.
- 5 Choose File | Save All to save your work.

Setting properties to connect the components

The next step is to connect the DataExpress components so that they can “talk” to each other by setting the appropriate component properties. You use the Properties tab of the Inspector to do this. The Inspector displays the properties for the component that is selected in the UI designer or in the component tree. For more information on using the Inspector, see the online help topic “Setting component properties in the Inspector.”

The first property you need to set is the `fileName` property of the `TextDataFile` component. This property tells JBuilder where to find the text file containing the data for `textDataFile1`.

- 1 Select `textDataFile1` in the component tree. The Inspector displays the properties for this component on the Properties tab.
- Note** You can only select DataExpress components in the component tree since they are non-visual components.
- 2 Click in the the edit area next to the `fileName` property in the Inspector (the property value field). It changes color to show that it is active for editing.
 - 3 Click the ellipsis button to display the File Name dialog box.
 - 4 Click the Browse button to display the Open dialog box, and browse to `/jbuilder/samples/DataExpress/TextDataFile/employee.txt`. Click Open.
 - 5 Click OK to close the File Name dialog box.
 - 6 Click the Source tab to view the resulting source code, which should look similar to:

```
textDataFile1.setFileName("/usr/local/jbuilder/samples/com/borland/  
samples/DataExpress/TextDataFile/employee.txt");
```

- Note** This tutorial will not work if you use a different text file than `employee.txt` at this time. This project also contains a SCHEMA file needed to import this text file. Later examples show you how to work with your own data files. See “Tutorial: Importing data from a text file” on page 10-2.

Now you need to connect `tableDataSet1` to the `textDataFile1` component.

- 1 Click the Design tab and select `tableDataSet1` in the component tree.
- 2 Click the `dataFile` property value, then click the down arrow and select the entry for `textDataFile1`.
- 3 Click the Source tab to see the source code generated for this:

```
tableDataSet1.setDataFile(textDataFile1);
```

- 4 Choose File | Save All to save your work.

The `employee.txt` file is designed to work with the default settings stored in the `TextDataFile` component for properties such as `delimiter`, `separator`, and `locale`. If it was designed differently, the appropriate properties could be set at this point.

Creating 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 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 the online help topics "Laying out your UI", and "Using layout managers" in *Building Applications with JBuilder*.

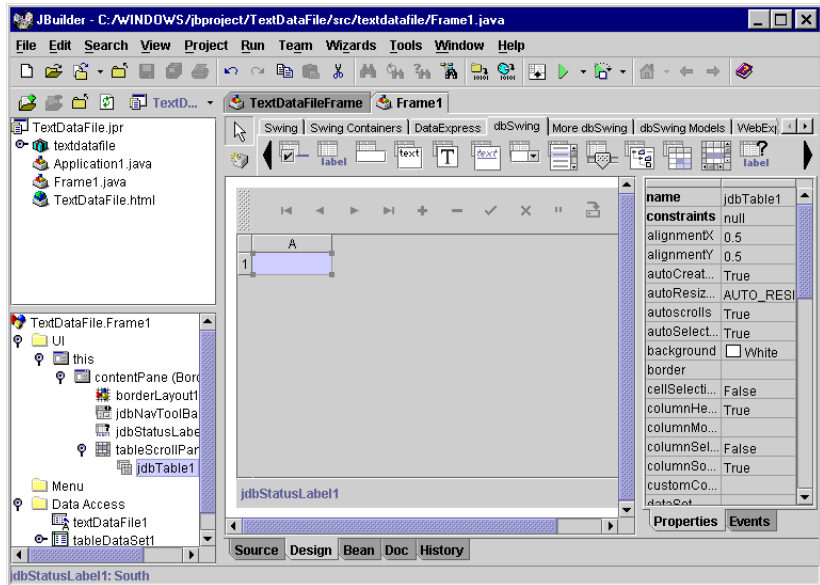
The steps below add the following UI components to the application from the `dbSwing` tab on the component palette:

- *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.

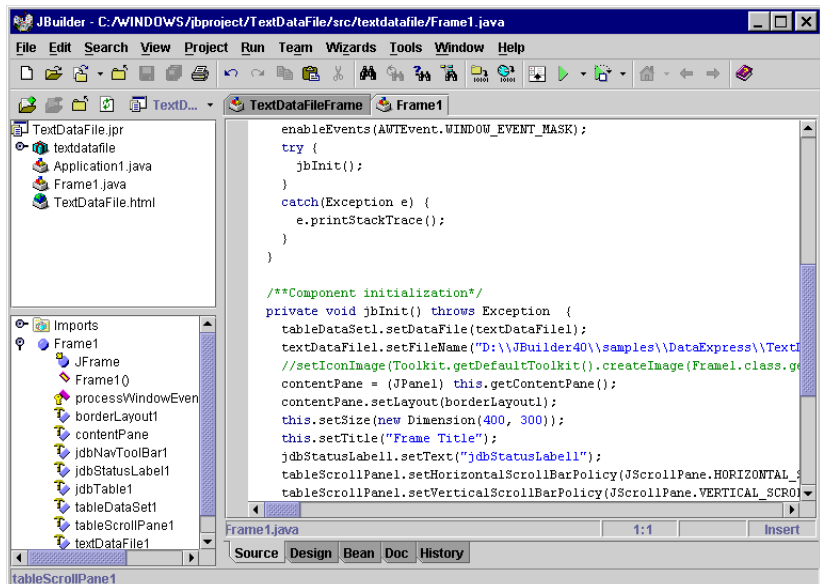
- 1 Click the Design tab on *Frame1.java* to open the UI designer, then click on *contentPane (BorderLayout)* in the component tree to select it. The UI designer displays black square sizing nibs around the selected component's outer edges in the UI designer.
- 2 Click the dbSwing tab on the component palette, then click the *JdbNavToolBar*.
- 3 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. *jdbNavToolBar1* automatically attaches itself to whichever *StorageDataSet* has focus.
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.
- 4 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.
- 5 Add a *JdbStatusLabel* component, using the same method. Drop it in the area near the center, bottom edge of the panel. *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.
- 6 Add a *TableScrollPane* component to the center of the panel. Make sure its *constraints* property is CENTER. A table should fill the rest of the panel.
Scrolling behavior is not available by default in any Swing component or dbSwing extension, so, to get scrolling behavior, you must add 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.
- 7 Finally, drop a *JdbTable* into the middle of the *tableScrollPane1* component in the designer. This fills the *tableScrollPane1* container with *jdbTable1*.

Your UI designer should look similar to this:



Note The scroll bars on a TableScrollPane are not visible in the designer because the default property setting for *vertical scroll bars* and *horizontal scroll bars* is AS_NEEDED. To display them all the time, change the settings for those properties to ALWAYS.

JBuilder creates the corresponding source code immediately for the elements you've added or modified in your application. To see this code, click the Source tab.



Connecting the DataExpress component to a UI component

The next step is to connect the `DataExpress` components to the UI components.

To connect the `DataExpress` components with the `JdbTable`, you must specify a `DataSet` in the `dataSet` property of the component.

To set the `dataSet` property of the `JdbTable` component and connect the UI component to live data,

- 1 Click the Design tab.
- 2 Select `jdbTable1` in the content pane.
- 3 Click the edit area beside the `dataSet` property in the Inspector.
- 4 Click the down arrow that appears.
- 5 Select `tableDataSet1` from the drop-down list. This list contains all instantiated `DataSet` components (of which there is only one in this example).

The column headers and live data appear in the table in the UI designer.

Compiling, running, and debugging your application

In the UI designer, data appears in the `JdbTable` and the application looks complete. But users will not be running your application in the JBuilder UI designer. So, the next step is to compile, run, test, and possibly debug the application.



To compile and run the application, click the Run Project button. Clicking the Run button compiles your source code (if it has not already been compiled). If no errors are found, a message indicating that the source code has successfully compiled displays in the message box at the bottom of the AppBrowser. The application UI displays in a new, separate window.

If there are syntax errors in the source code, the program is not automatically run. If syntax errors are found, the error messages appear in the message pane at the bottom of the AppBrowser. Double-click on the error message to locate the source of the error in the code. Errors in programming logic may not appear until run time when the application performs in unexpected ways.

You will probably not encounter any syntax errors in this simple application because all properties were selected from drop-down lists or browsers. Syntax errors are most likely to occur when you modify the generated code, or when you add in Java code manually.

If you would like more information on compiling and debugging applications in JBuilder, see the online help topics “Compiling Java programs” and “Debugging Java programs” in *Building Applications with JBuilder*.

You’ll notice the following behavior in the running application:

- The `JdbTable` displays the data from the text file.
- The `JdbStatusLabel` along the bottom of your application displays the number of rows in the table, and your current row position. The `JdbStatusLabel` displays messages generated by the `TableDataSet`. For example, when you change data in the table, the status label displays “Editing row”. As you move through the rows, you’ll notice the `JdbStatusLabel` updates automatically.
- Scroll bars are visible because there is more data available than will fit into the window.
- The `JdbNavToolBar` buttons at the top of your application enable you to move through the data, edit the data, or insert or delete rows. This occurs because the table and the toolbar are bound to the same `TableDataSet`. When two or more components are both bound to the same `DataSet`, they are said to “share” a cursor because they automatically synchronize to point to the same row of data.

Although the `JdbNavToolBar` has a Save button, this button is dimmed, and you cannot save changes to file-based data sources such as `employee.txt`. If this application had connected to a true database, the Save button provides a default mechanism for saving changes back to a data source. For more information on saving changes to a database, see “Saving changes back to your data source” on page 8-1. For more information on saving changes when the data source is a text file, see “Exporting data” on page 10-5.

Alternatively, the keyboard can be used to navigate the data in the table.

Summary

The application created for this tutorial reads data from a text file, displays the data in a `JdbTable` for viewing and editing, displays status messages to a `JdbStatusLabel`, and includes a `JdbNavToolBar` component to help browse though the data.

This tutorial was intended to familiarize you with the JBuilder environment and the basic requirements for developing a database application with JBuilder. Other topics in this chapter address retrieving data from various data sources using SQL queries and stored procedures. Chapter 10, “Importing and exporting data from a text file” provides more information on retrieving data from text files. Chapter 13, “Filtering,

sorting, and locating data” shows how to include additional database functions in your application.

See also

- “Retrieving data from a data source” on page 5-1
- “Querying a database” on page 5-14
- “Using parameterized queries to obtain data from your database” on page 5-27
- Chapter 6, “Using stored procedures”
- Chapter 10, “Importing and exporting data from a text file”
- Chapter 13, “Filtering, sorting, and locating data”

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” on page 5-19.

Property	Effect
<code>database</code>	Specifies what <code>Database</code> connection object to run the query against.
<code>query</code>	A SQL statement (typically a SELECT statement).
<code>parameters</code>	An optional <code>ReadWriteRow</code> from which to fill in parameters, used for parameterized queries.
<code>executeOnOpen</code>	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.
<code>loadOption</code>	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 5-27.
- **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 9-3.

Tutorial: Querying a database using the JBuilder UI

The following tutorial shows how to retrieve data using a `QueryDataSet` component. This example also demonstrates how to attach the resulting data set to a `JdbTable` for data viewing and editing.

Note We strongly recommended that before starting this tutorial you take the beginning database tutorial, “Tutorial: An introduction to JBuilder database applications” on page 5-4, to become familiar with using the visual design tools.

The finished example for this tutorial is available as a completed project in the `/samples/DataExpress/QueryProvider` directory of your JBuilder installation.

Retrieving data by querying a database

To create the application and retrieve data from a table,

- 1 Select **File | Close All**, then **File | New**.
- 2 Double-click the **Application** icon and accept all defaults to create a new application. `Frame1.java` will be opened by default in the content pane.
- 3 Select the **Design** tab to activate the UI designer.
- 4 Click the **Database** component on the **Data Express** tab of the component palette, then click anywhere in the UI designer or the component tree to add the component to the application. `database1` is added to the **DataExpress** folder in the component tree, and selected by default.
- 5 Click in the `connection` property value field in the Inspector, then click the ellipsis button to open the **Connection** property editor for `database1`.
- 6 Set the `connection` properties to the `JDataStore` sample **EMPLOYEE** table, as follows:

Property Name	Value
Driver	<code>com.borland.datastore.jdbc.DataStoreDriver</code>
URL	Use the Browse button to browse to <code>/jbuilder/samples/JDataStore/datastores/employee.jds</code> on your system, then click Open .
Username	Enter your name
Password	not required

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**.

You can view the code generated by the designer for this step by selecting the Source tab and looking for the `ConnectionDescriptor` code. Click the Design tab to continue.

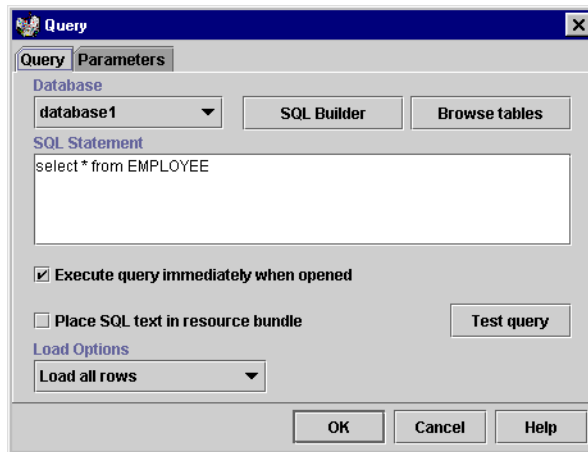
For more information on connecting to databases, see Chapter 4, “Connecting to a database.”



- 7 Now add a `QueryDataSet` component to your application from the Data Express tab of the component palette.
- 8 Click in the `query` property value field in the Inspector for `queryDataSet1`, then click the ellipsis button to open the Query property editor.
- 9 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 area beneath the button indicates `Success`, as shown below, click OK to close the dialog.



- 10 Switch to the More dbSwing tab on the component palette and add a `DBDisposeMonitor` to the application. This component will close the `DataStore` when the window is closed.
- 11 Set the `dataAwareComponentContainer` property for `dbDisposeMonitor1` to 'this'.
- 12 Choose File | Save All.

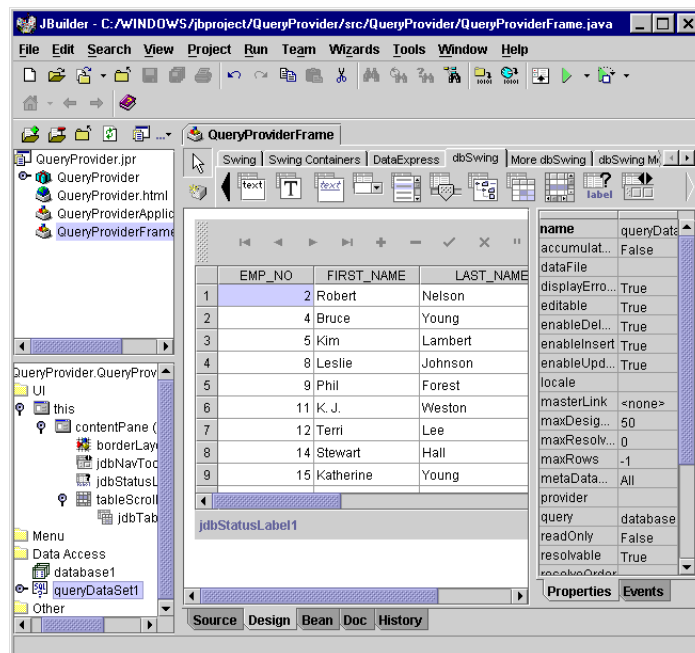
Creating the UI

Now create the UI for viewing and navigating the data in your application. Select the dbSwing tab on the component palette, and do the following:

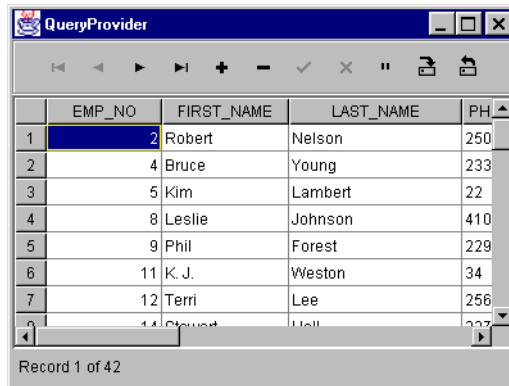
- 1 Select `contentPane` (`BorderLayout`) in the component tree. (Black sizing nibs around the edges of the panel in the designer show it is selected.)
- 2 Drop a `JdbNavToolBar` component into the designer at the top, center of the panel and set its constraints property to `NORTH`.
- 3 Drop a `JdbStatusLabel` component into the designer at the bottom, center of the panel and set its constraints property to `SOUTH`.
- 4 Drop a `TableScrollPane` component into the designer into the center of the panel, and set its constraints property to `CENTER`.
- 5 Drop a `JdbTable` component into the center of `tableScrollPane` and set its `dataSet` property to `queryDataSet1`.

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

The application looks like this in the designer:



- 6 Select Run | Run Project to run the application and browse the data set. The application looks like this when it is running:



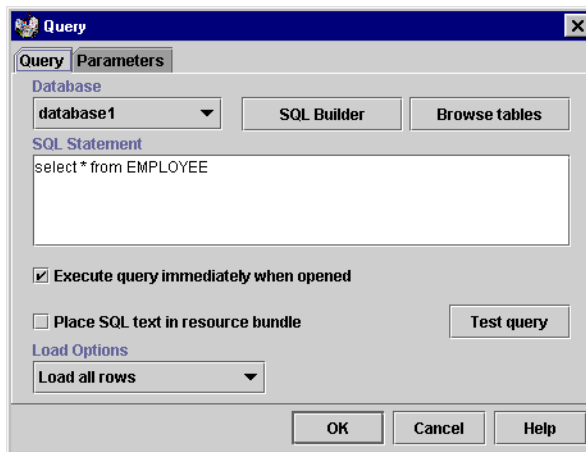
The screenshot shows a window titled "QueryProvider" with a toolbar and a table. The table has columns for EMP_NO, FIRST_NAME, LAST_NAME, and PH. The first row is highlighted, showing EMP_NO 2, FIRST_NAME Robert, LAST_NAME Nelson, and PH 250. The status bar at the bottom indicates "Record 1 of 42".

	EMP_NO	FIRST_NAME	LAST_NAME	PH
1	2	Robert	Nelson	250
2	4	Bruce	Young	233
3	5	Kim	Lambert	22
4	8	Leslie	Johnson	410
5	9	Phil	Forest	229
6	11	K. J.	Weston	34
7	12	Terri	Lee	256
8	14	Christ	Hell	227

To save the changes back to the data source, you can use the Save Changes button on the toolbar component or, for more control on how changes will be saved, create a custom data resolver, as described in the topic Chapter 8, "Saving changes back to your data source."

Setting properties in the query dialog

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.



The screenshot shows a dialog box titled "Query" with two tabs: "Query" and "Parameters". The "Query" tab is active, showing a "Database" dropdown set to "database1", "SQL Builder" and "Browse tables" buttons, and an "SQL Statement" text area containing "select * from EMPLOYEE". There are checkboxes for "Execute query immediately when opened" (checked) and "Place SQL text in resource bundle" (unchecked). A "Test query" button is also present. At the bottom, there are "OK", "Cancel", and "Help" buttons.

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

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.

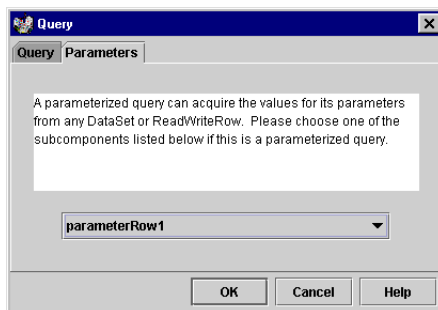
```
SELECT * FROM employee
```

- 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:
 - 1 Load All Rows: load all data up front.
 - 2 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.
 - 3 Load As Needed: load the rows as they are needed.
 - 4 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 5-22 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.

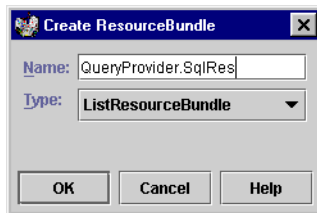


Any `ReadWriteRow`, such as `ParameterRow`, `DataSet`, and `DataRow` may be used as query or procedure parameters. In a `ParameterRow`, columns can simply be set up with the `addColumn`s and `setColumn`s methods. `DataSet` and `DataRow` should only be used if they already contain the columns with the wanted data. See “Using parameterized queries to obtain data from your database” on page 5-27 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:



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

pressed, 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 = ResourceBundle("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 selecting Help | Java 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 updateable

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 5-25 for information on doing this.

- **Set the `loadOption` property on the `Query/ProcedureDataSets` 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 `DataStore` 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,
 - 1 Set the `Resolver` property to a `QueryResolver`.
 - 2 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 your `Database`'s `useTransactions` property 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 `Connection`'s `commit()` method 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, right-click it, and select `Activate Designer`.
- 2 Press 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 7-4. To set the properties without using the designer,

- 1 Set the `StorageDataSet`'s `metaUpdate` property to `NONE`.
- 2 Set the `StorageDataSet`'s `tableName` property to the table name for single table queries.
- 3 Set the `Column`'s `rowID` property 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 `Column`'s `visible` or `hidden` property.
- 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 `Column`'s `tableName` property set for multi-table queries.
- 7 Set the `Column`'s `serverColumnName` property set 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 updateable

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

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

To make any data set updateable, 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 5-25 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 updateable. 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 the topics Chapter 2, “Understanding JBuilder database applications” and “Querying a database” on page 5-14 if you have not already done so. This topic is specific to parameterized queries.

In addition to the “Tutorial: Parameterizing a query” on page 5-27, the following parameterized query topics are discussed:

- “Using parameters” on page 5-33
- “Re-executing the parameterized query with new parameters” on page 5-35
- “Parameterized queries in master-detail relationships” on page 5-35

Tutorial: Parameterizing a query

The following tutorial 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 recommended that before starting this tutorial you take the beginning database tutorial, called “Tutorial: An introduction to JBuilder database applications” on page 5-4, to become familiar with using the visual design tools.

A completed version of this tutorial is available in the sample project `ParameterizedQuery.jpr`, located in the `/samples/DataExpress/ParameterizedQuery` directory of your JBuilder installation.

Creating the application

To create the application,

- 1 Select `File | Close All`.
- 2 Select `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 `Data Express` tab 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 in the `connection` property value in the `Inspector`.

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

Property name	Value
Driver	<code>com.borland.datastore.jdbc.DataStoreDriver</code>
URL	Browse to <code>/jbuilder/samples/JDataStore/datastores/employee.jds</code> in the local URL field.
Username	Enter your name
Password	not required

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`

Using parameterized queries to obtain data from your database

components to change the value in the `ParameterRow` so that the query can be refreshed using these new values.

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

Property name	Value
<code>columnName</code>	<code>low_no</code>
<code>dataType</code>	<code>INT</code>
<code>default</code>	<code>15</code>

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.

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

Property name	Value
<code>columnName</code>	<code>low_no</code>
<code>dataType</code>	<code>INT</code>
<code>default</code>	<code>50</code>

Adding a QueryDataSet

- 1 Add a `QueryDataSet` component from the Data Express tab to the application.
- 2 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
<code>Database</code>	<code>database1</code>
<code>SQL Statement</code>	<code>select emp_no, first_name, last_name from employee where emp_no >= :low_no and emp_no <= :high_no</code>

- 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 Click the Query tab. 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 `DataStore` when the window is closed.
- 8 Set the `dataAwareComponentContainer` property for the `DBDisposeMonitor` to 'this'.

Add the UI components

The instructions below assume you have taken 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` tab of the component palette. 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` tab into the center of `tableScrollPane` component. Set its `dataSet` property to `queryDataSet1`.
You'll notice that the table in the designer displays live data.
- 3 Select 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 tab.
- 2 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 `tableScrollPane` which is currently occupying the entire UI panel.
- 3 Make sure its `constraints` property is set to `NORTH`. (If `tableScrollPane` suddenly shrinks, check that its `constraints` property is still set to `CENTER`.)

- 4 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.
- 5 Drop a `JdbTextField` component from the `dbSwing` tab into `jPanel1`. This component holds the minimum value.
- 6 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.

- 7 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.
- 8 Add a `JLabel` from the `Swing` tab to `jPanel1`. This label will identify `jdbTextField1` as the minimum field.
- 9 Click on `jLabel1` in the UI designer and drag it to just above `jdbTextField1`.
- 10 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.
- 11 Add another `JdbTextField` and `JLabel` to `jPanel1` for the maximum value. Drag this pair of components to the right side of the panel.
- 12 Set the `columns` property for `jdbTextField2` to `10`, and its `text` property to `50`.
- 13 Set the `text` property for `jLabel2` to `Maximum value`, and expand its width to show all the text.
- 14 Align all four components as follows:

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.

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

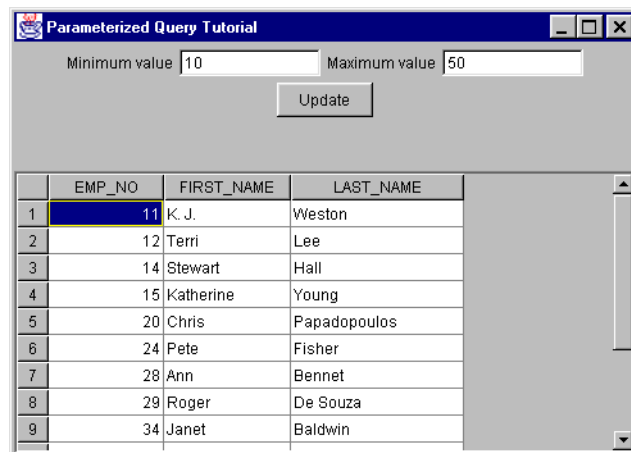
Clicking this button will update the results of the parameterized query with the values entered into the minimum and maximum value entry fields.

- 16** 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();
    }
}
```

- 17** 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 press 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 press 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 8-2 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` and `setColumns` 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 class hierarchy associated with the `DataSet` methods is as follows:

```

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 columns in the current row
- `ReadRow` methods give read access to columns 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 the "Tutorial: Parameterizing a query" on page 5-27, 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 5-35.

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 “Tutorial: Parameterizing a query” on page 5-27 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

Database application development is a feature of JBuilder Professional and Enterprise. Distributed application development is a feature of JBuilder Enterprise.

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
<code>database</code>	Specifies what Database connection object to run the procedure against.
<code>procedure</code>	A Java String representation of a stored procedure escape sequence or SQL statement that causes a stored procedure to be executed.
<code>parameters</code>	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> .
<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: <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 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 “Example: Using parameters with Oracle PL/SQL stored procedures” on page 6-10 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, select Tools | Database Pilot and refer to its online help.

The following topics related to stored procedure components are covered:

- Tutorial: Retrieving data using stored procedures
- Discussion of stored procedure escape sequences, SQL statements, and server-specific procedure calls
- Example: using InterBase stored procedures
- Example: using parameters with Oracle PL/SQL stored procedures
- Using Sybase stored procedures

Tutorial: Retrieving data using stored procedures

This tutorial shows how to provide data to an application using JBuilder's UI designer and a `ProcedureDataSet` component. This example also demonstrates how to attach the resulting data set to a `JdbTable` and a `JdbNavToolBar` for data viewing and editing.

The finished example for this tutorial may be available as a completed project in the `/samples/DataExpress/SimpleStoredProcedure` directory of your JBuilder installation. Other sample applications referencing stored procedures on a variety of servers are available in the `/samples/DataExpress/ServerSpecificProcedures` directory, and a sample of providers is available in the `/samples/DataExpress/CustomProviderResolver` directory.

Creating tables and procedures for the tutorial

These steps run a stored procedure that creates a table and insert, update, and delete procedures on the InterBase server (make sure have followed the setup instructions in "Setting up InterBase and InterClient" on page 3-4). This procedure is written in the InterBase language. These procedures will be used both in this section and in the "Tutorial: Saving changes using a QueryResolver" on page 8-6 and "Tutorial: Saving changes with a ProcedureResolver" on page 8-9.

- 1 InterBase Server and InterServer should be running on the same machine, unless it has been turned off.
- 2 Select File | Close All from the menu to close existing projects.
- 3 Select File | Open and open the project `ProcedureSetUp.jpr`, which is located in the `/jbuilder/samples/DataExpress/SimpleStoredProcedure/ProcedureSetup` directory of your JBuilder installation. If the project is not available or if you would like to explore the `CreateProcedures.java` file, see the section "Creating tables and procedures for the tutorial manually" on page 6-8.
- 4 Select Project | Properties.
- 5 Select the Required Libraries tab. Select InterClient. This option will be available if you have set it up as in "Adding a JDBC driver to JBuilder" on page 3-7.
- 6 Double-click `CreateProcedures.java` in the project pane and edit the path to the InterBase `employee.gdb` file to be that on your computer. (Use forward slashes in the path.)
- 7 Save the file, then right-click `CreateProcedures.java` in the project pane, and select Run. This step creates the tables and procedures on the server.

- 8 Select Tools | Database Pilot to verify that the tables and procedures are created.
- 9 Select File | Close Project from the menu.

Adding the DataSet components

To create this application and populate a data set using the stored procedure,

- 1 Select File | New and double-click the Application icon. Accept all defaults, or modify the path and project name to make them more descriptive.
- 2 Select Project Properties, and click the Required Libraries tab.
- 3 Add InterClient. This option will be available if you have set it up as in “Adding a JDBC driver to JBuilder” on page 3-7.
- 4 Close the dialog.
- 5 Select the Design tab to activate the UI designer
- 6 Select the Database component on the Data Express tab of the component palette, and click anywhere in the component tree.
- 7 Open the connection property editor for the Database component by selecting the connection property ellipsis button in the Inspector. Set the connection properties to the InterBase sample tables by setting the properties as indicated in the following table. These steps assume you have completed “Setting up InterBase and InterClient” on page 3-4.

Property name	Value
Driver	interbase.interclient.Driver
URL	jdbc:interbase://<IP address or localhost>/<path to .gdb file>
Username	SYSDBA
Password	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 text indicates Success, click OK to close the dialog.

The code generated by the designer for this step can be viewed by selecting the Source tab and looking for the ConnectionDescriptor. Select the Design tab to continue.

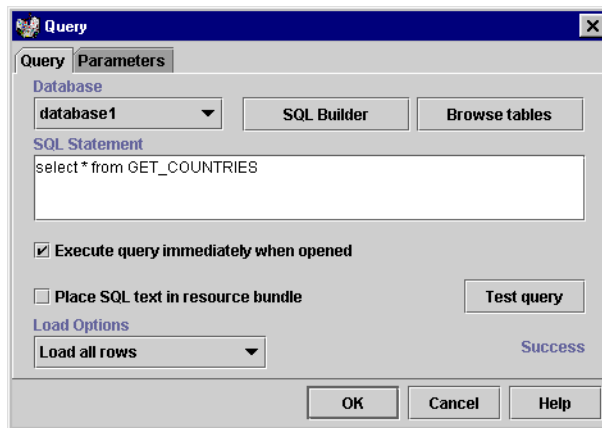
- 8 Place a `ProcedureDataSet` component from the Data Express tab of the component palette on the content pane. Set the `procedure` property of the `ProcedureDataSet` as follows:

Property name	Value
Database	database1
Stored Procedure Escape or SQL Statement	SELECT * FROM GET_COUNTRIES

Several procedures were created when `CreateProcedures.java` was run. The procedure `GET_COUNTRIES` is the one that will return a result set. The `SELECT` statement is how a procedure is called in the InterBase language. The other procedures will be used for resolving data in the topic “Tutorial: Saving changes with a `ProcedureResolver`” on page 8-9.

Tip You can use the Browse Procedures button in future projects to learn what stored procedures are available. See “Discussion of stored procedure escape sequences, SQL statements, and server-specific procedure calls” on page 6-7 for more information.

Click Test Procedure to ensure that the procedure is runnable. When the gray area beneath the button indicates `Success` as shown below, click OK to close the dialog.



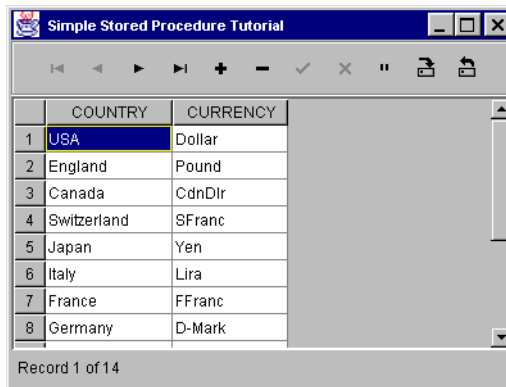
The code generated by this step is can be viewed by selecting the Source tab and looking for `setProcedure`. Click the Design tab to continue.

Adding visual components

This topic shows how to create a UI for your application using `dbSwing` components.

- 1 Select `contentPane (BorderLayout)` in the component tree.

- 2 Click on the `JdbNavToolBar` component on the `dbSwing` tab of the component palette, and drop the component in the area at the top of the panel in the UI designer. Set its `constraints` property to `NORTH`. `jdbNavToolBar1` automatically attaches itself to whichever `DataSet` has focus.
 - 3 Drop a `JdbStatusLabel` in the area at the bottom of the panel in the UI designer and set its `constraints` property to `SOUTH`. `jdbStatusLabel1` automatically attaches itself to whichever `DataSet` has focus.
 - 4 Drop a `TableScrollPane` component from the `dbSwing` tab to the center of the panel in the UI designer. Make sure its `constraints` property is set to `CENTER`.
 - 5 Select `tableScrollPane1` and drop a `JdbTable` into the center of it. Set its `dataSet` property to `procedureDataSet1`.
 - 6 Select `Run | Run Project` to run the application and browse the data set.
- The running application looks like this:



By default, the Save Changes button on the toolbar will save changes using a default `QueryResolver`. To customize the resolving capability in your application in order to edit, insert, and delete data in the running application, see

- “Tutorial: Saving changes using a `QueryResolver`” on page 8-6
- “Tutorial: Saving changes with a `ProcedureResolver`” on page 8-9

Stored procedures: Hints & tips

This set of topics includes tips to help you

- Understand the options for using a stored procedure
- Create the procedures used in the tutorial

Discussion of stored procedure 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.

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 "Example: Using parameters with Oracle PL/SQL stored procedures" on page 6-10. For an example using InterBase stored procedures, see "Example: Using InterBase stored procedures" on page 6-10.

Creating tables and procedures for the tutorial manually

Stored procedures consist of a set of SQL statements. These statements can easily be written and compiled in JBuilder by creating a Java file, entering the statements, then compiling the code. If you do not have access to the sample project SimpleStoredProcedure or if you would like to learn how to create a table and insert, update, and delete procedures from JBuilder, follow these steps:

- 1 Select File | Close All from the menu.
- 2 Select File | New Project.
- 3 Change the file directory and project name to SimpleStoredProcedure/ProcSetUp/ProcSetUp.jpr in the Project Wizard.
- 4 Select File | New, then select Class.
- 5 Change the Class Name to ProcSetUp in the Class wizard. Click OK to create the file ProcSetUp.java.
- 6 Edit the code in the Source window or copy and paste from online help to match the code below:

```
package ProcSetUp;

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

public class CreateProcedures {

    public static void main(String[] args) throws DataSetException {
        Database databasel = new Database();
        databasel.setConnection(new ConnectionDescriptor("jdbc:interbase:
//<IP address or localhost>/<path to .gdb file>", "SYSDBA",
"masterkey", false, "interbase.interclient.Driver"));
        try { databasel.executeStatement("DROP PROCEDURE GET_COUNTRIES"); }
        catch (Exception ex) {};
        try { databasel.executeStatement("DROP PROCEDURE UPDATE_COUNTRY"); }
        catch (Exception ex) {};
        try { databasel.executeStatement("DROP PROCEDURE INSERT_COUNTRY"); }
        catch (Exception ex) {};
        try { databasel.executeStatement("DROP PROCEDURE DELETE_COUNTRY"); }
        catch (Exception ex) {};
        databasel.executeStatement(getCountriesProc);
        databasel.executeStatement(updateProc);
        databasel.executeStatement(deleteProc);
        databasel.executeStatement(insertProc);
        databasel.closeConnection();
    }
}
```

```

static final String getCountriesProc =

"CREATE PROCEDURE GET_COUNTRIES RETURNS (           /r/n"+
"  COUNTRY VARCHAR(15),                             /r/n"+
"  CURRENCY VARCHAR(10) ) AS                         /r/n"+
"BEGIN                                              /r/n"+
"  FOR SELECT c.country, c.currency                 /r/n"+
"  FROM country c                                   /r/n"+
"  INTO :COUNTRY,:CURRENCY                          /r/n"+
"  DO                                              /r/n"+
"  BEGIN                                           /r/n"+
"    SUSPEND;                                       /r/n"+
"  END                                             /r/n"+
"END;";

static final String updateProc =

"CREATE PROCEDURE UPDATE_COUNTRY(                   /r/n"+
"  OLD_COUNTRY VARCHAR(15),                         /r/n"+
"  NEW_COUNTRY VARCHAR(15),                         /r/n"+
"  NEW_CURRENCY VARCHAR(20) ) AS                    /r/n"+
"BEGIN                                              /r/n"+
"  UPDATE country                                   /r/n"+
"    SET country = :NEW_COUNTRY                     /r/n"+
"    WHERE country = :OLD_COUNTRY;                  /r/n"+
"END;";

static final String insertProc =

"CREATE PROCEDURE INSERT_COUNTRY(                   /r/n"+
"  NEW_COUNTRY VARCHAR(15),                         /r/n"+
"  NEW_CURRENCY VARCHAR(20) ) AS                    /r/n"+
"BEGIN                                              /r/n"+
"  INSERT INTO country(country,currency)           /r/n"+
"    VALUES (:NEW_COUNTRY,:NEW_CURRENCY);         /r/n"+
"END;";

static final String deleteProc =

"CREATE PROCEDURE DELETE_COUNTRY(                   /r/n"+
"  OLD_COUNTRY VARCHAR(15) ) AS                     /r/n"+
"BEGIN                                              /r/n"+
"  DELETE FROM country                              /r/n"+
"    WHERE country = :OLD_COUNTRY;                  /r/n"+
"END;";
}

```

- 7 Right-click `ProcSetUp.java` in the project pane, then select Run. This step creates the tables and procedures on the server.
- 8 Select File | Close from the menu.

This is a very simple procedure. For tips on writing more complex stored procedures, consult your database documentation.

Stored procedures: InterBase, Oracle, and Sybase specific information

This set of topics includes tips to help you use

- InterBase stored procedures
- Oracle PL/SQL stored procedures
- Sybase stored procedures

Example: 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 `StatCODEent` 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 or see "Creating tables and procedures for the tutorial manually" on page 6-8 for an example of a stored procedure written in InterBase.

Example: 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 rcMyTable;
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` from the Data Express tab to the content pane.
- 5 Select the procedure property to bring up the `ProcedureDescriptor` dialog.
- 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. 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_procxmode` to change the transaction mode to either “anymode” or “unchained”. For more details, see the Sybase documentation.

Browsing sample applications that use stored procedures

In the `/samples/DataExpress/ServerSpecificProcedures` directory of your JBuilder installation, you can browse a sample application with sample code for 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[] columns, boolean updateColumns, boolean keepExistingColumns, boolean emptyRows)` method are explained below.

- `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 6-13. For more discussion on obtaining actual data, see “Obtaining actual data” on page 6-14.

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, select `/samples/DataExpress/QueryProvider/QueryProvider.jpr` from your JBuilder installation.
- 2 Double-click the file `QueryProvideFrame.java` file and click the Design tab from the bottom of the right pane of the AppBrowser.
- 3 Right-click the `queryDataSet1` object in the content pane, select `Activate Designer`. This displays the Column designer for the data set in the Design window. The Column designer looks like this for the `EMPLOYEE` sample table:

Column	columnName	dataType	preferredOr...	editMask	default
INTERNAL...	INTERNALRC	LONG	-1		
EMP_NO	EMP_NO	SHORT	-1		
FIRST_NA...	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_COU...	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 Select the Properties button, and select the `min` property to display in the Column designer.
- 2 Scroll to the `min` column, enter today's date for the `HIRE_DATE` field.
- 3 Press *Enter* to change the value.

To close the Column designer, select any UI component in the content pane, or right-click a different component, and select Activate Designer. In other words, the only way to close one designer is to open a different one.

See the topic "Ensuring data persistence" on page 14-23 for more information on using the Column designer.

The 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.

Table 7.1 RowIterator Generator dialog

Option	Description
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

Pressing 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 5-25

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

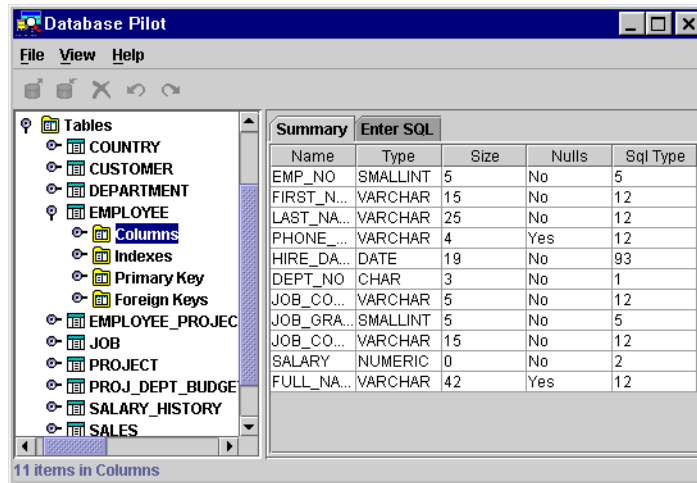
metadata-related properties named above. However, other properties, like `editMask` will not be touched.

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

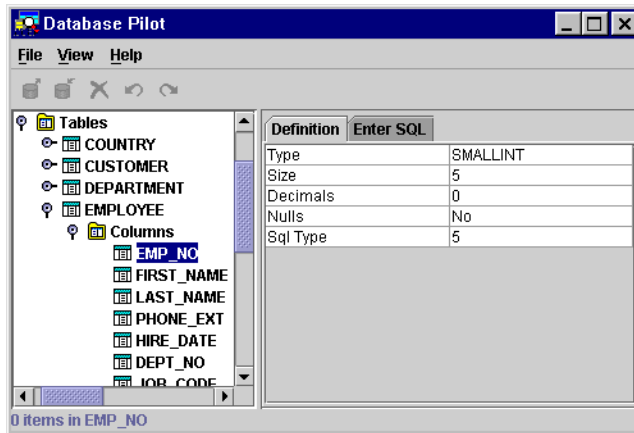
The Database Pilot is an all-Java, hierarchical database browser that also allows you to edit data. It presents JDBC-based meta-database information in a two-paneled window. The left pane contains a tree that hierarchically displays a set of databases and its associated tables, views, stored procedures,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

To display the Database Pilot, select `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.



For more information on using the Database Pilot, see its online help.

Optimizing a query

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 "Tutorial: Querying a database using the JBuilder UI" on page 5-16 for an example.
- 2 Open the UI designer by double-clicking the `Frame` container object in the project pane, and then clicking the `Design` tab in the `AppBrowser`.
- 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 the “Tutorial: Querying a database using the JBuilder UI” on page 5-16 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(s) 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 14-23. 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

- Need 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 7-7.

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 7-6 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

- 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

Database application development is a feature of JBuilder Professional and Enterprise.

Distributed application development is a feature of JBuilder Enterprise.

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 8-2 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 9-10.

- “Saving changes back to your data source with a stored procedure” on page 8-5 covers resolving changes made to a `ProcedureDataSet` back to its data source.
- “Resolving data from multiple tables” on page 8-11 provides the necessary settings for resolving changes when a query involves more than one table.
- “Using DataSets with RMI (streamable data sets)” on page 8-14 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 8-16 describes how to set custom resolution rules using the `QueryResolver` component and resolver events.
- “Exporting data” on page 10-5 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.

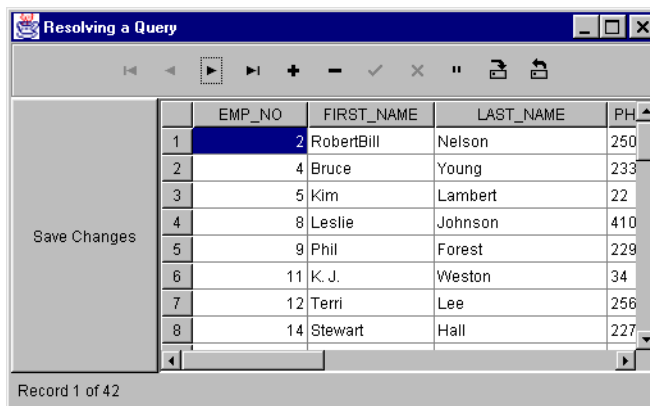
This topic explores the basic resolver functionality provided by the DataExpress package. It extends the concepts explored in the “Tutorial: Querying a database using the JBuilder UI” on page 5-16, to the resolving phase where you save your changes back to the data source.

To step through this tutorial, use the files you created from the “Querying a database” tutorial, or start with the completed sample files, located in the /samples/DataExpress/QueryProvider directory.

The “Querying a database” tutorial explored the providing phase where data is obtained from a data source. The tutorial instantiated a QueryDataSet and associated UI components, and displayed 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 pressed, the changes made to the data in the QueryDataSet are saved to the employee data file using the QueryDataSet’s default QueryResolver.

Tutorial: Adding a button to save changes from a QueryDataSet

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



To create this application,

- 1 Open the project file you created for the “Querying a database” tutorial by choosing File | Open, and then browsing to that project. If you did not complete the tutorial, you can access the completed project files from the /samples/DataExpress/QueryProvider directory of your JBuilder installation.

Note You should make backup copies of these files before modifying them since other tutorials in this book use the “Querying a database” files as a starting point.

- 2 Select the Frame file in the content pane.
- 3 Add a JButton component from the Swing tab of the component palette. Set the button’s text property to Save Changes. (See the finished application at the beginning of this tutorial 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 AppBrowser 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 choosing either,

- the Save Changes button you created, or
- the Save button of the `JdbNavToolBar`



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.

See also

“Customizing the default resolver logic” on page 8-16

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 “Using stored procedures” on page 6-1 by exploring the different methods for saving data changes back to a data source.

The topic “Tutorial: Retrieving data using stored procedures” on page 6-3 explores how to run a stored procedure in order to retrieve data. The tutorial creates a table on the server, and then creates insert, update, and delete procedures that will provide information on how to resolve changes back to the source. Using JBuilder’s IDE, the tutorial instantiates a `ProcedureDataSet` component and associated UI components, and displays the data returned from the `JDataStore` in a table. The Save button on the `JdbNavToolBar` can be used to save data changes back to the employee file when certain properties have been set.

In this topic, 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 “Tutorial: Saving changes using a `QueryResolver`” on page 8-6.
- 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 “Tutorial: Saving changes with a `ProcedureResolver`” on page 8-9.

Tutorial: 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 tutorial shows how.

The finished example for this tutorial may be available as a completed project in the `/samples/DataExpress/SimpleStoredProcedure/` directory of your JBuilder installation. Other sample applications referencing stored procedures on a variety of servers are available in the `/samples/DataExpress/ServerSpecificProcedures/` directory.

To complete the application and save changes back to the COUNTRY table,

- 1 Select File | Close if you have any open projects. Select File | Open. Open the project file you created for “Tutorial: Retrieving data using stored procedures” on page 6-3. We will add resolving capability to the existing project.

At this point in the tutorial, you can run the application and view and navigate data. In order to successfully insert, delete, or update records, however, you need to provide more information to the `QueryResolver`, as follows. The `QueryResolver` is invoked by default unless a `ProcedureResolver` is defined (see “Tutorial: Saving changes with a `ProcedureResolver`” on page 8-9). Then proceed with the following steps:

- 2 Select `Frame1.java` in the project pane. Select the Design tab to activate the UI designer.
- 3 Select `procedureDataSet1` in the component tree.

- 4 Set the `tableName` property of `procedureDataSet1` to "COUNTRY" in the Inspector.
- 5 Verify that the `resolvable` property of the `procedureDataSet1` is set to `True`.
- 6 Click the expand icon to the left of `procedureDataSet1` in the project pane to expose the columns of the data set.
- 7 Select the key column, which is named COUNTRY.
- 8 Set the `rowID` property of the COUNTRY column to `True`.
- 9 Select Run | Run Project to run the application.

The application compiles and displays in a separate window. Data is displayed in a table with the toolbar and a status label that reports the current row position and row count. You can now insert, update, or delete records and save the changes back to your database.

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.

In the above example, you could add a `JButton` coded to handle saving changes in place of the `JdbNavToolBar`. For more detailed information about how to do this, see "Tutorial: Adding a button to save changes from a `QueryDataSet`" on page 8-3. With the button control selected in the component tree, select the Event tab of the Inspector, select the `actionPerformed()` method, double-click its value field, and add the following code in the Source window:

```
try {
    database1.saveChanges(procedureDataSet1);
    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.

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.

A example of code that uses this method of resolving data to a database follows in “Tutorial: Saving changes with a `ProcedureResolver`” on page 8-9. In the case of `InterBase`, also see “Example: Using `InterBase` stored procedures with return parameters” on page 8-11.

Tutorial: Saving changes with a ProcedureResolver

The following tutorial 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.

To complete the application and save changes back to the `COUNTRY` table with custom defined insert, update, and delete procedures, first, open the project file you created for "Tutorial: Retrieving data using stored procedures" on page 6-3. Resolving capability will be added to the existing project.

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 tab 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:
 - 1 Select `procedureResolver1` in the component tree and click its `deleteProcedure` property in the Inspector.
 - 2 Double-click in the `deleteProcedure` property value field to bring up the `DeleteProcedure` dialog.
 - 3 Set the `Database` property to `database1`.
 - 4 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
```

5 Edit this statement to be:

```
execute procedure DELETE_COUNTRY :COUNTRY
```

See the text of the procedure in “Creating tables and procedures for the tutorial manually” on page 6-8 or 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:

- 1** Select, then double-click the `insertProcedure` property of the `ProcedureResolver` to open the `insertProcedure` dialog.
- 2** Set the `Database` field to `database1`.
- 3** Click `Browse Procedures`, then double-click the procedure named `INSERT_COUNTRY`.
- 4** 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:

- 1** Select, then double-click the `updateProcedure` property of the `ProcedureResolver` to open the `updateProcedure` dialog.
- 2** Set the `Database` property to `database1`.
- 3** Click `Browse Procedures`, then double-click the procedure named `UPDATE_COUNTRY`.
- 4** 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 Select `Run | Run 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 updateable?

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 5-14.

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

For a multi-table query, one of the tables can be updateable 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 QueryDataSets` or `ProcedureDataSets` 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 JBuilder `/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`.

A tutorial describing how to write a custom `ProcedureResolver` is available in the “Tutorial: Saving changes with a `ProcedureResolver`” on page 8-9.

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 5-14 for how to do this.
- 2 Select the `Frame` file in the content pane. Click the `Design` tab to display the UI designer.
- 3 Click the `QueryResolver` component from the `Data Express` tab 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.jpr` 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 `DataSetViews` after the data has been resolved, or if an exception occurs during resolution. If the `DataSetViews` 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 of 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 `DataSets` 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 9-10.

Establishing a master-detail relationship

Database application development is a feature of JBuilder Professional and Enterprise. Distributed application development is a feature of JBuilder Enterprise.

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 8-11.

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
- Fetching details
- Editing data in master-detail data sets
- Steps to creating a master-detail relationship
- Tutorial: Creating a master-detail relationship
- Saving changes in a master-detail relationship

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 8-11.

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).

Be careful when using the `cascadeUpdates` and `cascadeDelete` options for master-detail relationships. When using these options, one row of a detail data set may be updated or deleted, but the others may not be. For example, an event handler for the `editListener`'s `deleting()` event 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 `cascadeDelete` options, see the `MasterLinkDescriptor` topic in the *DataExpress Component Library Reference*.

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 - 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`, there should be a `WHERE` clause that defines the relationship of the detail columns in the current `QueryDataSet` 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 5-35.

When the 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

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.

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.

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, or go to a tutorial that uses the sample database files shipped with JBuilder, “Tutorial: Creating a master-detail relationship” on page 9-6.
- 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 9-3 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 will not display the linked columns.

To save changes back to the tables, see “Saving changes in a master-detail relationship” on page 9-10.

Tutorial: Creating a master-detail relationship

This tutorial 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 file, and one that selects all of the employees. This tutorial 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 Select File | Close. Select File | New, and double-click the Application icon. Accept all defaults.
- 2 Select the Design tab in the content pane.
- 3 Select a Database component from the Data Express tab of the component palette, click in the component tree or the UI designer to add the component to your application.
- 4 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 Chapter 3, “Setting up

JBuilder for database applications” (which includes adding the DataStore library to the project):

Property name	Value
Driver	com.borland.datastore.jdbc.DataStoreDriver
URL	Browse to your local copy of /jbuilder/samples/JDataStore/datastores/employee.jds
Username	Enter your name
Password	not required

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.

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.

- 5 Select a `QueryDataSet` component from the Data Express tab, 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	database1
SQL Statement	select * from COUNTRY

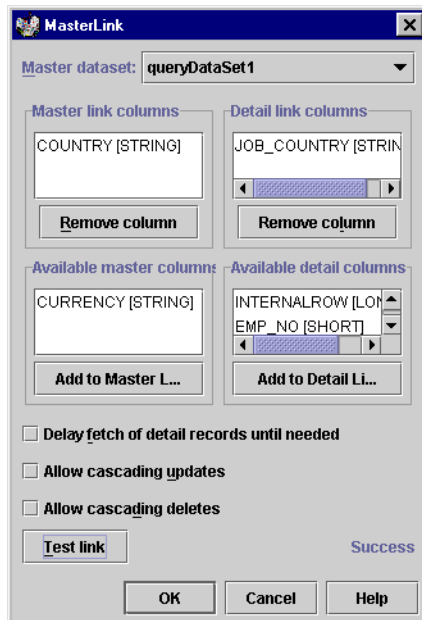
Click the Test Query button to ensure that the query is runnable. When the status area indicates `Success`, click OK to close the dialog.

- 6 Add another `QueryDataSet` component to your application. Select its `query` property in the Inspector. This will set up the query for the detail data set. In the `query` custom property editor, set the following properties:

Property name	Value
Database	database1
SQL Statement	select * from EMPLOYEE

Click the Test Query button to ensure that the query is runnable. When the status area indicates `Success`, click OK to close the dialog.

- 7 Select the `masterLink` property for the detail data set (`queryDataSet2`) in the Inspector. In the `masterLink` property editor, set the properties as follows:
 - The `Master DataSet` property provides a choice 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`.
 - 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.
 - 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.
 - 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 9-3.
 - Click `Test Link`. The dialog should look like below when you are done. When successful, click `OK`.



- 8 Add a `DBDisposeMonitor` to your application from the More `dbSwing` tab. The `DBDisposeMonitor` will close the `DataStore` when the window is closed.
- 9 Set the `DBDisposeMonitor`'s `dataAwareComponentContainer` property to 'this'.

To create a UI for this application,

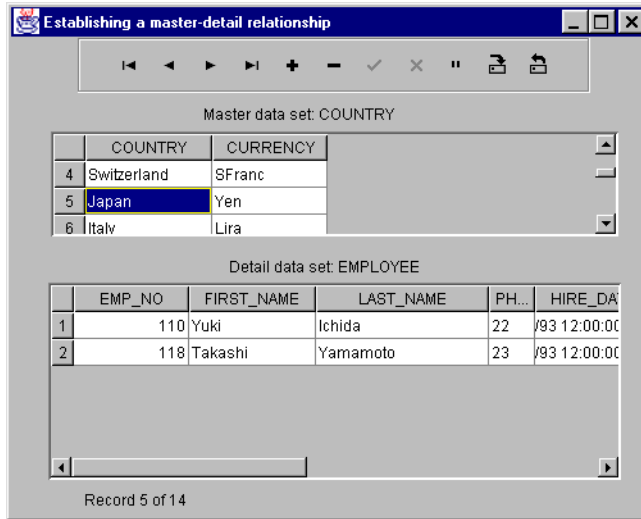
- 1 Select `contentPane` (`BorderLayout`) in the component tree. Set its `layout` property to `null`.
- 2 Add a `JdbNavToolBar` component from the `dbSwing` tab. Drop the component in the area at the top of the panel in the UI Designer. `JdbNavToolBar` automatically attaches itself to whichever `DataSet` has focus.
- 3 Add a `JdbStatusLabel` and drop it in the area at the bottom of the panel in the UI designer. `JdbStatusLabel` automatically attaches itself to whichever `DataSet` has focus.
- 4 Select a `TableScrollPane` component from the `dbSwing` tab, 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. 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 selecting `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:



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 “Saving changes back to your data source” on page 8-1, and the options for exporting data to a text file are discussed in “Exporting data” on page 10-5.

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 8-2.

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 9-11 for more information.

- Place a `QueryResolver` in your application to customize resolution. See “Customizing the default resolver logic” on page 8-16 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 `QueryDataSets` 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. 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.

Importing and exporting data from a text file

Database application development is a feature of JBuilder Professional and Enterprise.

Distributed application development is a feature of JBuilder Enterprise.

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` file. This topic discusses the first two options. Importing data using an existing `.SCHEMA` file is discussed in “Tutorial: An introduction to JBuilder database applications” on page 5-4. Your text file has a valid `.SCHEMA` file only if it has previously been exported by JBuilder.

These are the topics covered:

- Tutorial: Importing data from a text file
- Adding columns to a `TableDataSet` in the editor
- Importing formatted data from a text file
- Retrieving data from a JDBC data source
- Exporting data

Tutorial: Importing data from a text file

This tutorial shows how to provide data to an application using a `TableDataSet` component and a comma-delimited text data file. This type of file can be exported from most desktop databases. This application is available as a finished project in `TextFileImportExport.jpr` in the `/samples/DataExpress/TextFileImportExport` directory of your JBuilder installation.

For this example, create a text file to import as follows:

- 1 Open a text editor.
- 2 Enter the following three rows and two columns of data (a column of integer values and a column of string values) into a blank text file. Press the *Enter* or *Return* key at the end of each row. Enter the quotation marks as well as the data.

```
1, "A"
2, "B"
3, "C"
```

- 3 Save the file with the name `ImportTest.txt`. Close the file.

To read the data from this text file and create a database application in JBuilder,

- 1 Select `File | Close`. Select `File | New`. Double-click the Application icon and accept all defaults.
- 2 Select the Design tab in the content pane.
- 3 Select a `TextDataFile` component from the Data Express tab of the component palette, and click in the component tree or the UI designer to add the component to your application. Select the following properties in the Inspector, and set their values as indicated:

Property name	Value
<code>delimiter</code>	" (double quote)
<code>separator</code>	, (comma)
<code>fileName</code>	<path to>ImportTest.txt

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 Select a `TableDataSet` component from the Data Express tab of the component palette, click in the component tree or UI designer to add the component to your application. Select its `dataFile` property, and set it to `textDataFile1`.
- 5 Add columns to the `TableDataSet`. 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 10-4. If you have completed this tutorial previously and exported the data to a text file, JBuilder created a `.SCHEMA` file that provides column definitions when the text file is opened and you do not need to add columns manually.

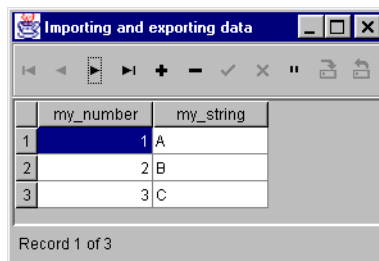
Click the expand icon to the left of the `TableDataSet` component to expose existing columns. In this case, there are no existing columns, so select `<new column>` and set the following properties in the Inspector for the first column:

- `dataType` to `SHORT`
 - `caption` and `columnName` to `my_number`
- 6 Set the properties for the second column by selecting `<new column>` again. Set the following properties in the Inspector:
 - `dataType` to `STRING`
 - `caption` and `columnName` to `my_string`

To make the data available to your application,

- 1 Add a `TableScrollPane` to the panel in the UI designer, then drop a `JdbTable` into the center of it.
- 2 Set the `dataSet` property of the `jdbTable1` to `tableDataSet1`. 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.
- 3 Select `Run | Run Project` to compile and run the application.

The running application looks like this, after adding file export capabilities, as discussed in “Tutorial: Exporting data to a text file” on page 10-6.



- 4 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 10-5 for more information on exporting.

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 “Tutorial: Importing data from a text file” on page 10-2. 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 “Edit/display mask patterns” 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 “Tutorial: Importing data from a text file” on page 10-2.

```
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 10-11.

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.

Tutorial: Exporting data to a text file

When you export data from a `TableDataSet` to a text file, JBuilder creates a `.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.

Building on the example in “Tutorial: Importing data from a text file” on page 10-2, this tutorial demonstrates how to use the UI designer to add a button for saving the data, with any changes, back to the same text file.

- 1 If you have not already done so, create the project in the “Tutorial: Importing data from a text file” topic. If you have created this project, open it now.
- 2 Select the Design tab of the content pane.
- 3 Select `contentPane` (`BorderLayout`) in the content pane and change its `layout` property to ‘null’ in the Inspector.
- 4 Select `tableScrollPane1` in the component tree. 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.

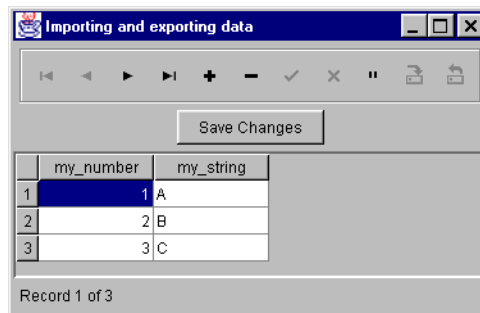
- 5 Add a `JButton` component from the Swing tab to the UI designer. On the Properties tab of the Inspector, set the `text` property to Save Changes.
- 6 Click the Events tab of the Inspector. Select, then double-click the `actionPerformed()` method. This changes the focus of the AppBrowser from the 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);
}
```

- 7 Run the application by selecting Run | Run Project.

When you run the application, if it compiles successfully, the application appears in its own window.



Data is displayed in a table, with a Save Changes button. Make and view changes as follows:

- 1 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. Save the changes back to the text file by clicking the Save Changes button.
- 2 View the resulting text file in a text editor. It will now contain the following data:

```
1, "Apple"
2, "B"
3, "C"
```

- 3 Close the text file.

JBuilder automatically creates a `.SCHEMA` file to define the contents of the text file.

- 4 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 = ISO8859_1
LOCALE = en_US
DELIMITER = "
SEPARATOR = ,
FIELD0 = my_number,Variant.SHORT,-1,-1,
FIELD1 = my_string,Variant.STRING,-1,-1,
```

- 5 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.

Tutorial: 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. Complete the example in “Tutorial: Exporting data to a text file” on page 10-6, close the running application, then complete the following steps in JBuilder. These steps demonstrate creating 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 locale syntax of your computer in the `my_date` column of the first row. For example, with the `locale` property set to English (United States), you must enter the date in a format of MM/dd/yy, like 11/16/95. 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", 11/16/95
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 = ISO8859_1
LOCALE = en_US
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 read and 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. In the running Frame window, enter a date in the my_date column of the first row using the format of the `exportDisplayMask` property, such as 11-16-1995. 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 = ISO8859_1
LOCALE = en_US
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`.

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 “Tutorial: Exporting data to a text file” on page 10-6. 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 8-16.

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 8-16.

Using data modules to simplify data access

Database application development is a feature of JBuilder Professional and Enterprise.

Distributed application development is a feature of JBuilder Enterprise.

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

Create the data module with the wizard

To create a data module,

- 1 Create a new project.
- 2 Select 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 the OK button to close the dialog. 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,

- 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,
 - 1 Select a `Database` component from the Data Express tab of the component palette.
 - 2 Click in the component tree or the UI designer to add the `Database` component to the `DataModule`.
 - 3 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,

- Double-click the data module file in the project pane to open it in the content pane.
- Select the Design tab of the content pane to activate the UI designer.
- Select the component to which you want to add business logic, then click the Events tab in the Inspector.
- 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 Select File | Save All. Note the name of the project, the package, and the data module.
- 2 Compile the data module class by selecting Run | Make Project. This creates the data module class files in the directory specified in Project | Project Properties, Output Path.
- 3 Select 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 file.

Select Project | Project Properties. Select Required Libraries from the Paths tab, 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. To do this,

- 1 Click Add.
- 2 Click New.
- 3 Enter the name for the library (like Employee Data Module).
- 4 Select the location where you want your <library name>.library file to go. You have a choice between JBuilder, Project, and User Home. 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 folder 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.

- 5 Click Add.
- 6 Browse to the folder which contains the path to the class file or archive you wish to add. JBuilder automatically determines the paths to class files, source files, and documentation within this folder.
- 7 Click OK.
- 8 Click OK.
- 9 Click OK.
- 10 At this point you should see your new library added to the list of required libraries.

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 Select File | New. Select Application. Enter the appropriate package and class information. (Optionally, open an existing project using File | Open).
- 2 Select the application's Frame file in the content pane.
- 3 Make sure Data Express is specified as one of the Required Libraries. If Data Express is not listed under Required Libraries in the Project Properties,
 - 1 Click Add.
 - 2 Select Data Express.
 - 3 Click OK until the Project Properties dialog is closed.
- 4 Import the package that the data module class belongs to (if it is outside your package) by selecting Wizards | Use Data Module.
- 5 Click the ellipsis to open the Select Data Module dialog. 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 the project compiled without errors and that it was properly added to the required libraries for the project.
- 6 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.

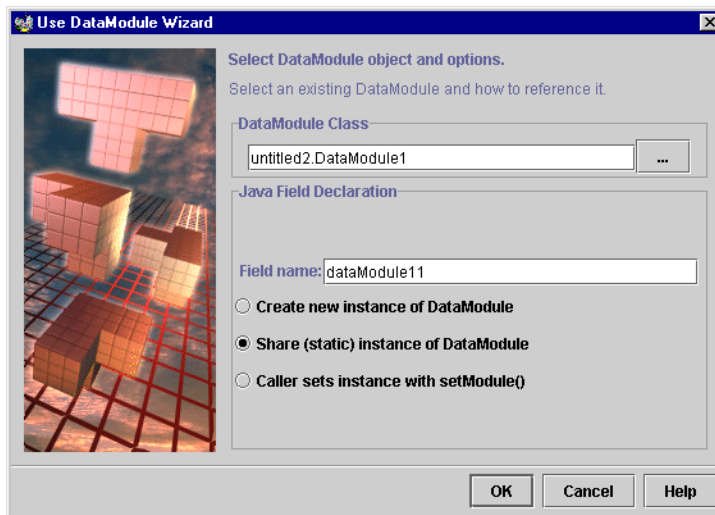
Click the Design tab to open the UI designer; the instance of the data module appears in the content pane. Clicking the entry for the data module does not display its `DataSet` components nor its `Column` components. This is intentional to avoid modification of the business logic contained in the data 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 Data Module dialog

When you select Wizards | Use Data Module, you will see the following dialog:



Select a data module by clicking the ellipsis in 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 package or archive to your libraries. 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`.

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 2-tier 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,

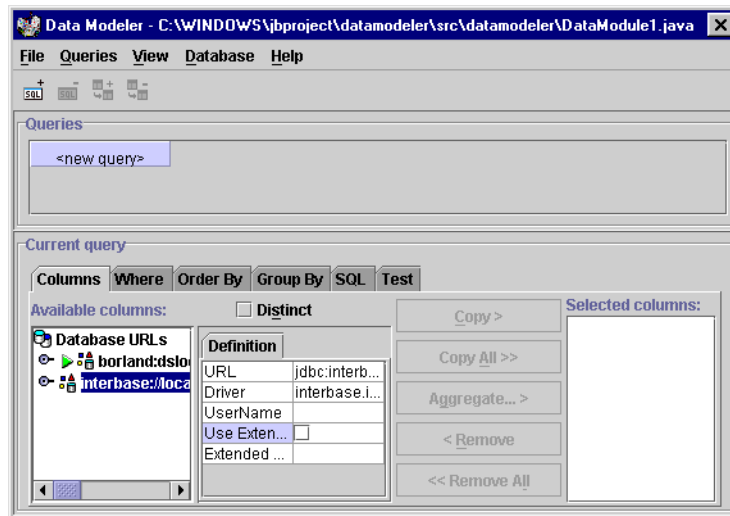
- 1 Choose File | New Project to start the Project wizard.

- 2 Choose a location and name for the project.
- 3 Click the Finish button.

For more specific information about creating projects, see the online help topic “Creating and managing projects.”

To display the Data Modeler,

- 1 Choose File | New.
- 2 Double-click the Data Module icon.
- 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.



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 an 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 */samples/JDataStore/datastores/employee.jds*. The *employee.jds* database is located under the *samples* directory of your JBuilder installation, which may be different on your system. 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 selecting columns you want to add to the query from a table, or by selecting an aggregate function that operates on a specific column.

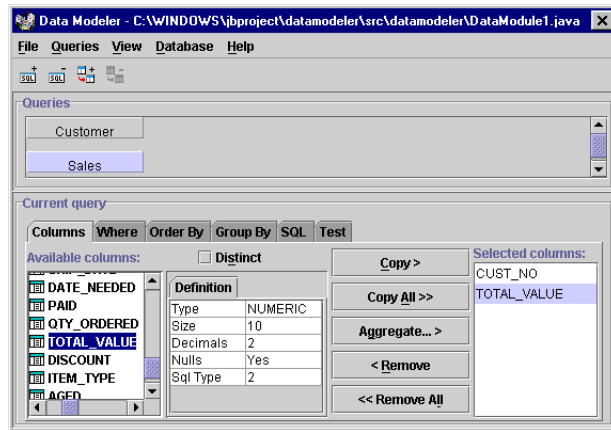
To view the tables, double-click the Tables node or choose the Tables expand icon.

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.

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.

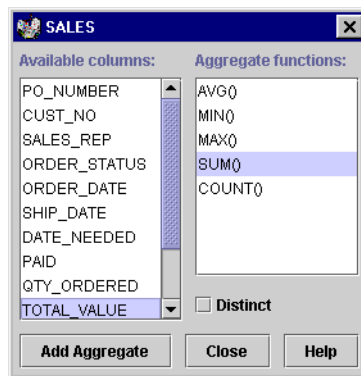
- The SELECT clause specifies the list of columns to be retrieved. To add one or more columns to a query's SELECT statement,
 - 1 Select a column you want to add from the table you want to access.
 - 2 Click the Copy button.

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.



- Aggregate functions provide a summary value based on a set of values. Aggregate functions include SUM, AVG, MIN, MAX, and COUNT. To add an aggregate function to the query,

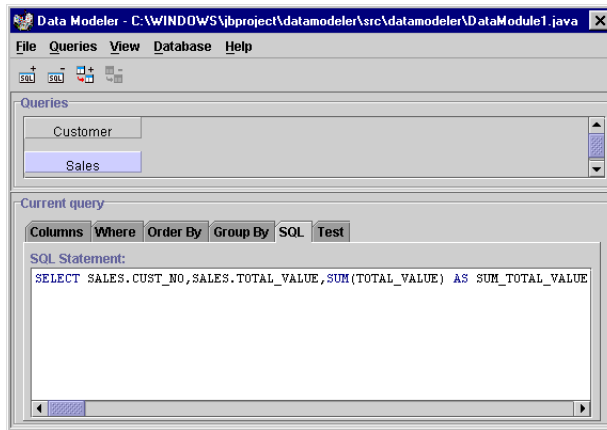
1 Click the Aggregate > button to display a dialog box.



- Click the column whose data values you want aggregated in the Available Columns list.
- Click the function you want to use on that column from the Aggregate Functions column.
- If you want the function to operate on only unique values of the selected column, check the Distinct check box.
- 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.” To view it, click the SQL tab.

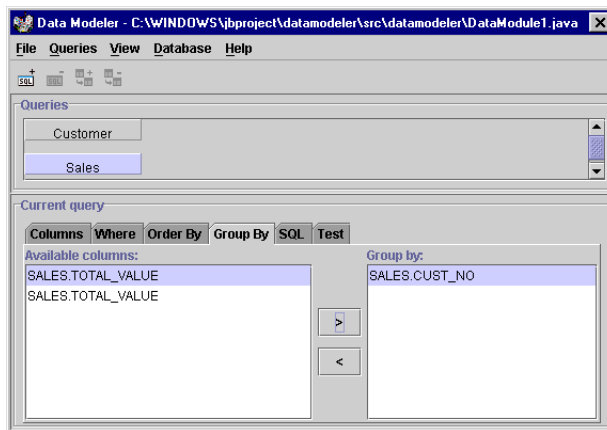


Adding a Group By clause

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.



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 > 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, only rows with unique values are returned. 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.



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.

To transfer a column as a column name to the Where Clause box, select a column in the Columns list and click the Paste Column button.

To transfer a column as a parameter as in a parameterized query, select a column in the Columns list and click the Paste Parameter button.

Select the operator you need in the Operators drop-down list and click the Paste button. Every Where clause requires at least one operator.

If your query requires a function, select the function you need in the Functions drop-down list 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'
```

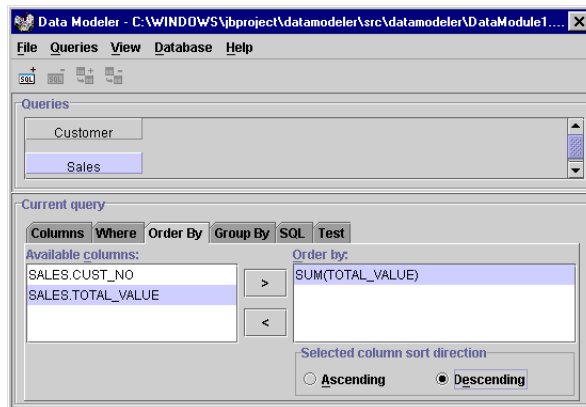
You would select and paste the COUNTRY column and the = operator. To complete the query, you would type in the data value directly, which in this case is 'USA'.

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.



- 3 Select the column you want the query sorted by in the Available Columns box and click the button with the > symbol on it 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.

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.

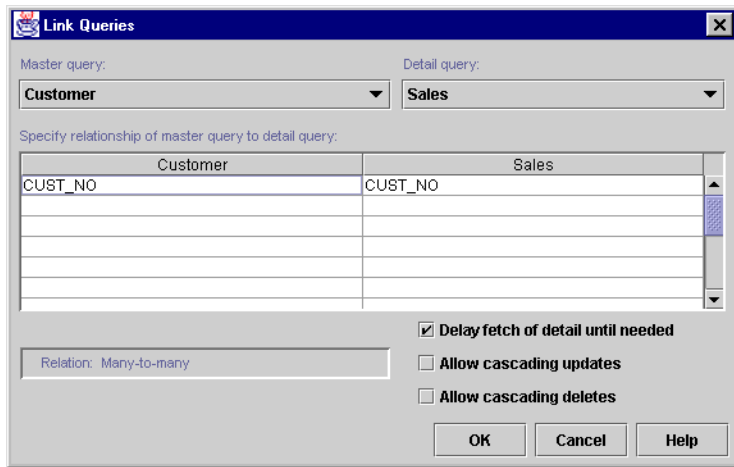
Building multiple queries

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:
 - Choose Queries | Link.
 - In the Queries panel, click-and-drag the mouse pointer from the query you want to be the master query to the one you want to be the detail query.

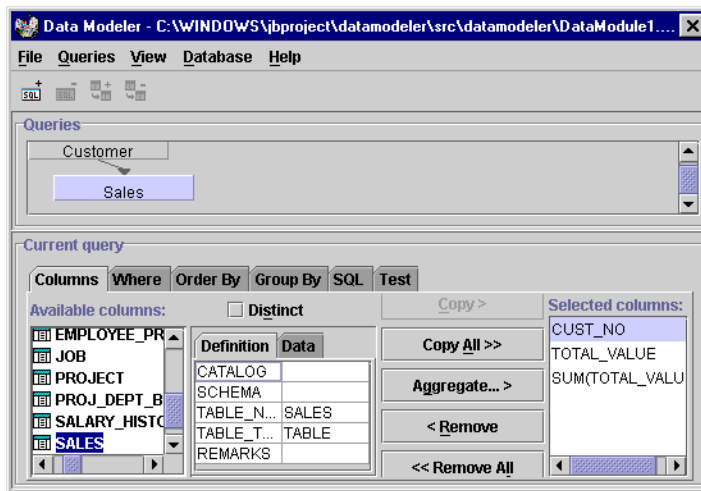


- 2 Select a query to be the master query in the Master Query list.
- 3 Select a query to be the detail query in the Detail Query list.

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:
 - 1 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.

- 2 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.
- 3 Choose 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.



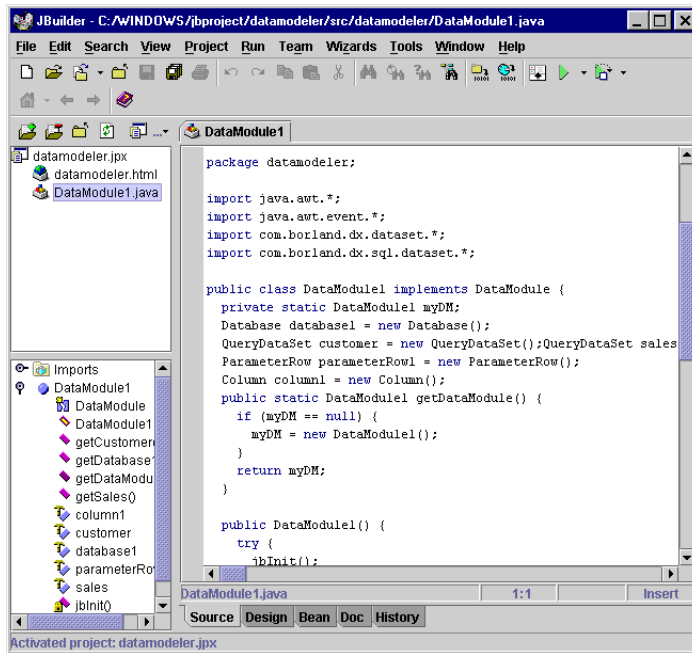
For more information about master-detail relationships, see Chapter 9, “Establishing a master-detail relationship.”

Saving your queries

To save the data module you built,

- 1 Choose File | Save in the Data Modeler and specify a name with a .java extension.
- 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.

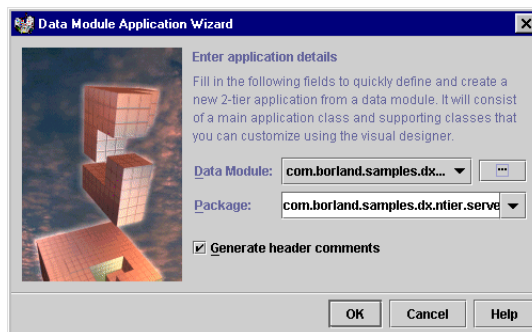


Generating database applications

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 2-Tier Application wizard icon in the object gallery:

- 1 Choose File | New and select the Applications tab.
- 2 Double-click the Data Module Application icon.



- 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 Choose 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 client2tier package:
 - 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.
- 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).
- 2 Choose Wizards | Use Data Module 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 Choose 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.datamodel.*;

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 Data Module 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();
    }
}

```


Persisting and storing data in a DataStore

Database application development is a feature of JBuilder Professional and Enterprise. Distributed application development is a feature of JBuilder Enterprise.

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

When to use a `DataStore`:

- **Organization.** To organize an application's `StorageDataSets`, 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.** DataStore foot print is very small. StorageDataSets also provide excellent data binding support for data-aware UI components.
- **Performance.** To increase performance and save memory for a large StorageDataSet. StorageDataSets 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.

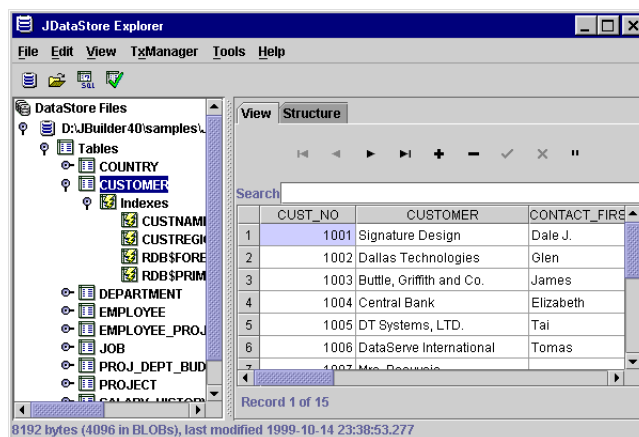
For more information on using *DataStores*, refer to the *JDataStore Developer's Guide*.

Using the DataStore Explorer

Using the DataStore 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 DataStore, 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 DataStore.
- 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 | DataStore Explorer menu command to launch the DataStore Explorer.



DataStore operations

To create a new DataStore,

- 1 Open the DataStore Explorer by selecting Tools | DataStore Explorer.
- 2 Select File | New or click the New DataStore 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 Select 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 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 10-5). 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 Choose OK.

To import a file into a file stream,

- 1 Select Tools | Import | File.
- 2 Supply an input file name and the store name of the data set to be created, and choose OK.

To verify the open DataStore,

- 1 Select Tools | Verify DataStore or click the Verify DataStore 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 selecting View | Verifier Log.

For more information on using the DataStore Explorer, refer to the *JDataStore Developer’s Guide*.

Filtering, sorting, and locating data

Database application development is a feature of JBuilder Professional and Enterprise.

Distributed application development is a feature of JBuilder Enterprise.

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 is created using the JBuilder AppBrowser 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 in order to have your application perform a particular task.

These tutorials assume that you are comfortable using the JBuilder environment and do not provide detailed steps on how to use the user interface. If you're not yet comfortable with JBuilder, refer to the "Tutorial: An introduction to JBuilder database applications" on page 5-4 or to the online help topic "Designing a user interface."

All of the following examples and tutorials involve accessing SQL data stored in a local database. These examples use the sample files in the downloadable Samples Pack, using the local `JDataStore` JDBC driver. For instructions on how to setup and configure JBuilder to use the sample `JDataStore` driver, see "Deploying database applications" on page 3-8.

We encourage you to use the samples as guides when adding these functions to your application. Finished projects and Java source files for many of these tutorials, with comments in the source file where appropriate, are provided. (If you downloaded JBuilder, you also need to download the Samples Pack.) All files referenced by these examples are found in the JBuilder samples directory. If you experience problems running the sample applications, see “JBuilder sample files” on page 3-2 for information critical to this process.

Note Some of the samples run only with JBuilder Enterprise.

To create a database application, you first need to connect to a database and provide data to a `DataSet`. “Retrieving data for the tutorials” on page 13-2 sets up a query that will be used for each of the following database tutorials. 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 data. Filtering temporarily hides rows in a `DataSet`.
- Sorting data. Sorting changes the order of a filtered or unfiltered `DataSet`.
- Locating data. Locating positions the cursor within the filtered or unfiltered `DataSet`.

Retrieving data for the tutorials

This topic provides the steps for setting up a basic database application that can be used with the tutorials in this chapter. The query that will be used in these tutorials is: `SELECT * FROM EMPLOYEE`

This SQL statement selects all columns from a table named `EMPLOYEE`, included in the sample `JDataStore`.

To set up an application for use with the tutorials,

- 1 Close all open projects (select File | Close Project).
- 2 Select File | New Project.
- 3 Enter a name and location for the project in the Project Wizard. Click Finish.
- 4 Select File | New from the menu. Double-click the Application icon.
- 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 Data Express tab of the component palette, then click in the component tree or the UI designer to add the component to the application.

Open the connection property editor for the `Database` component by selecting, then clicking the connection property ellipsis 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
Password	not required

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 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.”

- 8 Add a `QueryDataSet` component to the designer by clicking on the `QueryDataSet` component on the Data Express tab and then clicking in the component tree or the UI Designer.

Select the `query` property of the `QueryDataSet` component in the Inspector, click its ellipsis to open the `QueryDescriptor` dialog, and set the following properties:

Property name	Value
Database	<code>database1</code>
SQL Statement	<code>SELECT * FROM EMPLOYEE</code>

Click Test Query to ensure that the query is runnable. When the status area indicates Success, click OK to close the dialog.

- 9 Add a `DBDisposeMonitor` component from the More `dbSwing` tab. The `DBDisposeMonitor` will close the `DataStore` when the window is closed.
- 10 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 as follows:

- 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` tab 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`.

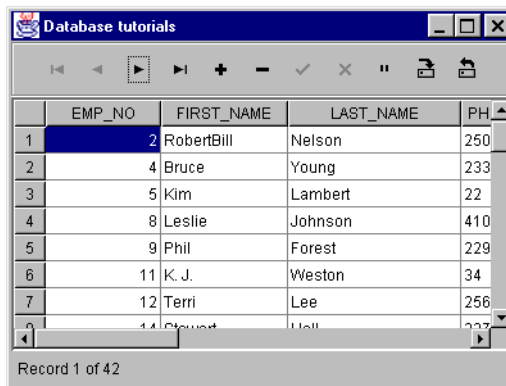
You'll notice that the designer displays live data at this point.

- 6 Select `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 tutorials 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:



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 14-22.

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 13-9.
- Locating positions the cursor within the filtered or unfiltered `DataSet`. For more information on locating data, see “Locating data” on page 13-14.

Tutorial: Adding and removing filters

This tutorial shows 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 tutorials” on page 13-2. 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` tab and a `JButton` component from the `Swing` tab. 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,

- 1 Select the Design tab.
- 2 Select the `queryDataSet1` in the component tree.
- 3 Select the Events tab of the Inspector.
- 4 Select the `filterRow` event.
- 5 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.

- 6 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,
 - 1 Select the Design tab.
 - 2 Select the `JButton` in the component tree.
 - 3 Click the Events tab on the Inspector.

- 4 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.

- 5 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 = jdbTextField1.getText();
        columnValue = jdbTextField2.getText();
        Column column = queryDataSet1.getColumn(columnName);
        formatter = column.getFormatter();

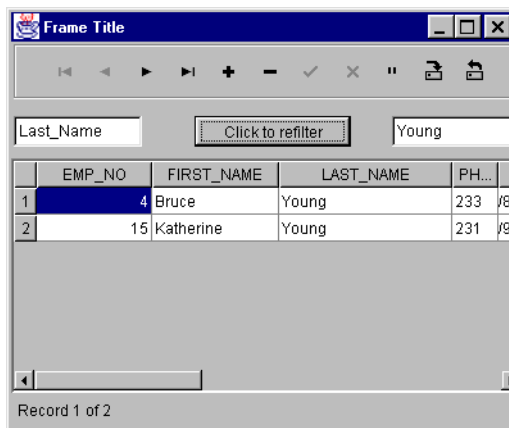
        // Trigger a recalc of the filters

        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:



To test this application,

Enter the name of the column you wish to filter (for example, `Last_Name`) in the first `JdbTextField`.

- 1 Enter the value you wish to filter for in the second `JdbTextField` (for example, `Young`).
- 2 Click the `JButton`.

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 dataset is sorted as follows:

- First on the primary column.
- The secondary column defined in the sort breaks any ties when columns defined in the primary sort are not unique.
- Subsequent columns defined in the sort continue to break ties.
- 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:

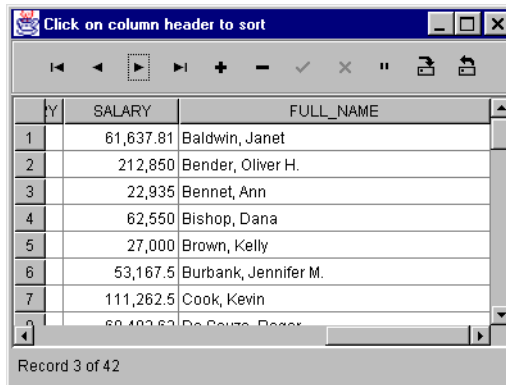
- Case sensitivity applies only when sorting data of type `String`.
- Case sensitivity applies to all `String` columns in a multi-column sort.
- Sort directions (ascending/descending) are set on a column-by-column basis.
- Null values sort to the top in a descending sort, to the bottom in an ascending sort.

Sorting and indexing data are closely related. See “Understanding sorting and indexing” on page 13-12 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.



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

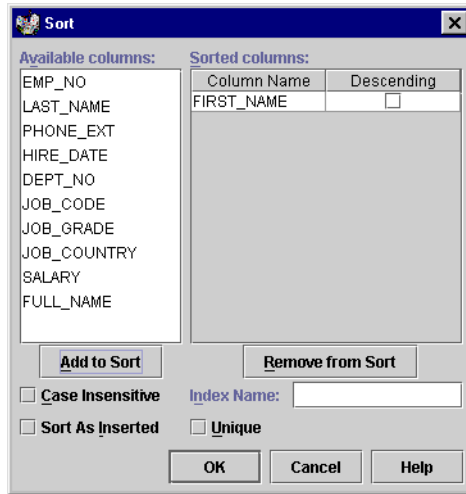
- View the columns that currently control sort order.
- Select from among the sortable columns in the `DataSet`.
- Add and remove selected columns to and from the sort specification.
- Set the case-sensitivity of the sort.
- Set the sort order to ascending or descending on a column-by-column basis.
- 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 tutorials” on page 13-2.
- 2 Click the Design tab. 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 will look like this:



- 6 Click the OK button.

The property values you specify in this dialog are stored in a `SortDescriptor` object.

- 7 Select Run | Run Project to compile and run the application. It will look like this:

The application window shows the following data:

	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	Dexter	De Souza	200

Record 16 of 42

Understanding sorting and indexing

There are two options on the Sort dialog 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` - only rows with unique values for the columns defined as `sortKeys` in the `SortDescriptor` can be added or updated in a `DataSet`.

What is a unique index?

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*, any duplicate rows are moved to another “duplicates” data set.

How this works: when the unique `sort` property is applied for the first time, 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.

- 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.
- 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.

- Index Name

Enter a name in this field to create a named index. This is the user-specified name to be associated with the sort specification (index) being defined in the dialog.

What is a named index?

- 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. *Maintained* means that each index is updated 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 intent of the index name is to let you revert to a previously defined sort. The index has been "maintained" (kept up-to-date), so that it can be re-used. And in fact, 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.
- 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.

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 the following:

```
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:

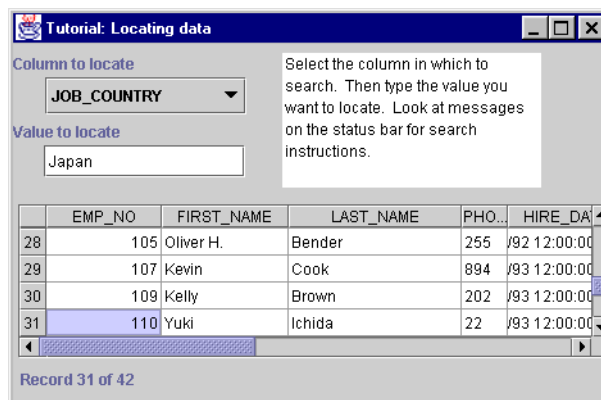
- An interactive locate using a `JdbNavField`, where the user can enter values to locate when the application is running.
- A locate where the search values are programmatically set.

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` subdirectory of `JBuilder` includes a finished example of an application that uses the `JdbNavField` under the project name `LocatingData.jpr`. 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:



To create this application,

- 1 Create a new application by following “Retrieving data for the tutorials” on page 13-2. 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 screen shot of the running application (shown above) for the approximate positioning of components.

- 2 Add a `JdbNavField` from the More dbSwing tab of the component palette to the UI designer. Set its `dataSet` property to `queryDataSet1`.
- 3 Add a `JdbComboBox` from the dbSwing tab 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 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 tab. Place it 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 label for search instruction.
```

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 tab. 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.

When you run the application, you'll notice the following behavior:

- Select the column name on which you want to perform the locate in the `JdbComboBox`.
- 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.

- Press the *UpArrow* or *DownArrow* keys to perform a “locate prior” or “locate next” respectively.
- The status label updates to reflect the current status of the application, for example,
 - Initially, the status area displays “Enter a value and press enter to begin search”.
 - If a value is not found in the table, the status area displays “Could not find a matching column value”.
 - 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 13-17.)
- 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 “Locate class variables” in the online help 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 `Locate` variables. The `Locate` class variables 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 13-17.) 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:

- `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 13-18.

Adding functionality to database applications

Database application development is a feature of JBuilder Professional and Enterprise.

Distributed application development is a feature of JBuilder Enterprise.

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 13, "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 AppBrowser 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.

These tutorials assume that you are comfortable using the JBuilder environment and do not provide detailed steps on how to use the user interface. If you're not yet comfortable with JBuilder, refer to the "Tutorial: An introduction to JBuilder database applications" on page 5-4 or to the online help topic "Designing a user interface."

All of the following examples and tutorials involve accessing SQL data stored in a local `JDataStore`. Finished projects and Java source files are provided in the JBuilder samples directory (`/samples/DataExpress`) for many of these tutorials, 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`. “Retrieving data for the tutorials” on page 13-2 sets up a query that can be used for each of the following database tutorials.

- Creating lookups, includes information on creating a lookup using a picklist.
- Using calculated columns
- Aggregating data with calculated fields
- Adding an Edit or Display Pattern for data formatting
- Presenting an alternate view of the data

Creating lookups

A `Column` can derive its values 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 picklist.

This topic covers providing values to a column using a picklist to enter a new value to a column, and it also covers creating a lookup that will display values from another column.

- “Tutorial: Data entry with a picklist” on page 14-3 discusses using a picklist to lookup a value in a column of another data set for **data entry**.

This type of lookup displays a list of choices in a drop-down list. The choices that populate the list come the unique values of a column of another data set. The tutorial gives the steps for looking up a value in a picklist 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 used provide values for the picklist. The choices will be available for data entry in a visual component, such as a table, when the application is running.

There are many other things you can do with a picklist. For more ideas, see the pick list sample in the `dbSwing` samples directory.

- “Tutorial: Creating a lookup using a calculated column” on page 14-5 discusses using a lookup field to **display values** from a column of another data set.

This type of lookup retrieves values from a specified table based on criteria you specify and displays it as part of the current table. In order 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 are not resolved to or provided from its data source, although they can 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 display a part description for display in an invoice line item or looking up a zip 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 13-14.

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.

Tutorial: Data entry with a picklist

This tutorial shows how to create a picklist 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. This project can be viewed as a completed application by running the sample project `Picklist.jpr`, located in the `/samples/DataExpress/Picklist` subdirectory of your `JBuilder` installation.

This application is primarily created in the designer.

- 1 Create a new application by following “Retrieving data for the tutorials” on page 13-2. 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 form the query to populate the list of choices. Set the `query` property of `queryDataSet2` as follows:

For this option	Make this choice
Database	database1
SQL Statement	select COUNTRY from COUNTRY

Click Test Query. When successful, click OK.

- 3 Click the expand icon to the left of the `queryDataSet1` component in the component tree to expose all of the columns. Select `JOB_COUNTRY`.
- 4 Open the `pickList` property editor in the Inspector to bring up the `pickListDescriptor`. 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>
Display Column?	checked
<code>queryDataSet1</code>	<code>JOB_COUNTRY</code>

Click OK.

- 5 Click the Source tab. Enter the following code after the call to `jbInit()`. 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.

```
queryDataSet2.open();
```

- 6 Run the application by selecting 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 picklist field

To remove a picklist,

- 1 Select the column that contains the picklist in the component tree.
- 2 Open the `pickListDescriptor` dialog by clicking in the `pickList` property in the Inspector.
- 3 Set the `PickList/Lookup DataSet` field to `<none>`.

Tutorial: Creating a lookup using a calculated column

This tutorial 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.jpr, located in the /samples/DataExpress/Lookup subdirectory of your JBuilder installation.

For more information on using the `calcFields` event to define a calculated column, refer to “Using calculated columns” on page 14-7.

- 1 Create a new application by following “Retrieving data for the tutorials” on page 13-2. 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	database1
SQL Statement	select * from EMPLOYEE_PROJECT

Click Test Query. When successful, click OK.

- 3 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.
- 4 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 manually edit the `setColumns()` method to change the

position of this or any column. 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.

- 5 Select the Events tab of the Inspector (assuming the new column is still selected in the content pane). 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.

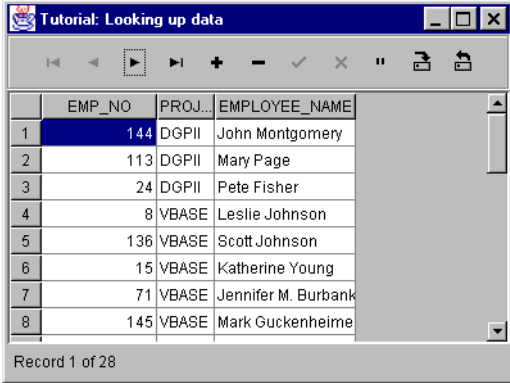
    lookupRow.setShort("EMP_NO", changedRow.getShort("EMP_NO"));
    if (queryDataSet1.lookup(lookupRow, resultRow,
        Locate.FIRST))
        calcRow.setString("EMPLOYEE_NAME",
            resultRow.getString("FIRST_NAME") +
            " " + resultRow.getString("LAST_NAME"));
    }
}
```

- 6 Click the Source tab. Enter the following code after the call to `jbInit()`. 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.

```
queryDataSet1.open();
```

- 7 Select `Run | Run Project` to run the application.

The running application will look like this:



	EMP_NO	PROJ...	EMPLOYEE_NAME
1	144	DGP11	John Montgomery
2	113	DGP11	Mary Page
3	24	DGP11	Pete Fisher
4	8	VBASE	Leslie Johnson
5	136	VBASE	Scott Johnson
6	15	VBASE	Katherine Young
7	71	VBASE	Jennifer M. Burbank
8	145	VBASE	Mark Guckenheime

Record 1 of 28

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.

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.

In order 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 are not resolved to or provided from its data source, although they can be written to a text file. For more information on defining a calculated column in the designer, see "Tutorial: Creating a calculated column in the designer" on page 14-8. 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 can be used to group and/or 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 14-10.

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 “Creating lookups” on page 14-2.

Values for all calculated columns in a row are computed in the same event call.

These are the topics covered:

- Tutorial: Creating a calculated column in the designer
- Aggregating data with calculated fields
- Tutorial: Aggregating data with calculated fields
- Setting properties in the AggDescriptor
- Creating a custom aggregation event handler

Tutorial: Creating a calculated column in the designer

This tutorial builds on the example in “Retrieving data for the tutorials” on page 13-2. 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 subdirectory of your JBuilder installation under the project name CalculatedColumn.jpr.

- 1 Create a new application by following “Retrieving data for the tutorials” on page 13-2. 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.

- Click the expand icon beside `queryDataSet1` in the component tree to display all columns. Select `<new column>`. 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 manually edit the `setColumns()` method to change the position of the new columns or any other persistent column. 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.

- 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.
- 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.

- Import the `java.math.BigDecimal` class to use a `BIGDECIMAL` data type. Add this statement in the Source window to the existing `import` statements.

```
import java.math.BigDecimal;
```

- 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 columns displays the value of (SALARY * 1.10). The running application looks like this:

	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 Souza, Roger	60,400.00	66,440.00

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 you can use to define 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 below. 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 tutorial section.

Tutorial: 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` subdirectory of your JBuilder installation.

- 1 Create a new application by following “Retrieving data for the tutorials” on page 13-2. 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 on `queryDataSet1` in the component tree. This forms the query to populate the data set with values to be aggregated. Open the `query` property of `queryDataSet1`, and modify the SQL Statement to read:

```
SELECT CUST_NO, PO_NUMBER, SHIP_DATE, TOTAL_VALUE from SALES
```

Click the Test Query button to test the query and ensure its validity. When successful, click OK.

- 3 Click on the expand icon beside `queryDataSet1` in the component tree. Select `<new column>`. In the Inspector, set the following properties:

Property name	Value
<code>caption</code>	<code>GROUP_TOTAL</code>
<code>columnName</code>	<code>GROUP_TOTAL</code>
<code>currency</code>	<code>True</code>
<code>dataType</code>	<code>BIGDECIMAL</code>
<code>calcType</code>	<code>aggregated</code>
<code>visible</code>	<code>Yes</code>

A new column is instantiated and the following code is add to the `jbInit()` method. To view the code, select the Source tab to view, 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);
```

- 4 Add a `JdbTextField` from the `dbSwing` tab of the component palette to the UI designer. Set its `dataSet` property to `queryDataSet1`. 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.

- 5 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.

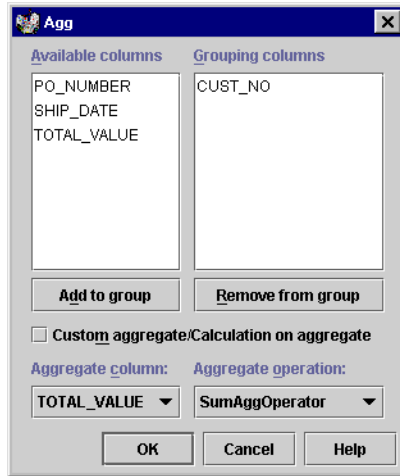
- 6 Select the `GROUP_TOTAL` column in the content pane. To define the aggregation for this column, double-click the `agg` property to display the `agg` property editor.

In the `agg` property editor,

- 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.
- Select `TOTAL_VALUE` from the `Aggregate Column` list to select this as the column that contains the data to be aggregated.
- 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.

When the `agg` property editor looks like the one below, click OK.

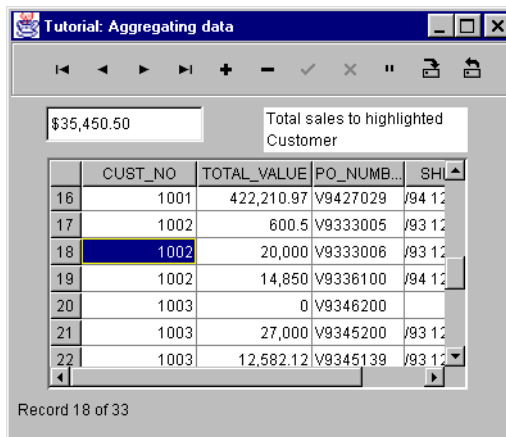


This step generates the following source code in the `jbInit()` method:

```
column1.setAgg(new com.borland.dx.dataset.AggDescriptor(new
    String[] {"CUST_NO"}, "TOTAL_VALUE", new
    com.borland.dx.dataset.SumAggOperator()));
```

- 7 Run the application by selecting `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 (i.e., 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 `AggOperators` provided with `DataExpress` package do 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 `AggOperators` built into `DataExpress` package as well as any user-defined `AggOperators` within the same class context as the `AggDescriptor`'s `Column`.

Users desiring to perform calculations on aggregated values (e.g., 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 `AggDescriptor`'s `aggOperator` property 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 `DataSets`. 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 as numbers, dates, etc., 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.jpr` file, which is in the `/samples/dbSwing/MultiLingual` subdirectory 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 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 the 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, and timestamp 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 the 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 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 <code>L</code> specifies any letter A-Z, entry required. The <code>0</code> (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 tutorials” on page 13-2. 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 Data Express tab to the component tree or the UI designer.
- 3 Set the `storageDataSet` property of the `DataSetView` component to `queryDataSet1`.
- 4 The `DataSetView` navigates independently of its associated `StorageDataSet`. Add another `TableScrollPane` and `JdbTable` to the UI designer. 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 Select the Frame file in the project pane. Select the Design tab.
- 2 Select the `DataSetView` component.
- 3 On the Properties page in the Inspector,
 - Select the `sort` property to change the order records are displayed in the `DataSetView`. See “Sorting data” on page 13-9 for more information on the `sortDescriptor`.
 - 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 13-5 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 in order 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.

To set a `Column` level property,

- 1 Open any project that includes a `DataSet` object, for example, select any project file (.jpr) in the /samples/DataExpress/ subdirectory of your JBuilder installation.
- 2 Double-click the `Frame` file 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:

Column	columnName	dataType	preferredOr...	editMask	default
INTERNAL...	INTERNALRC	LONG	-1		
EMP_NO	EMP_NO	SHORT	-1		
FIRST_NA...	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_COU...	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 Column designer, the columns for that data set are displayed in a table in the UI designer. A toolbar for adding, deleting, navigating, and restructuring 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 Restructure button is only available if the data set's `store` property has been set to a `DataStore` property. For more information on `DataStore`, see Chapter 12, "Persisting and storing data in a `DataStore`."

Restructure compiles the `this` component and launches a separate VM to perform a restructure of the `DataStore` associated with the data set. While the Restructure is running, a dialog box is displayed to show the status of the restructure and to allow you to cancel the restructure.

- 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 7-4.
- The Make All MetaData Dynamic button enables you to update a query after the table may have changed on the server. To do this, you must first make the metadata dynamic, then persist it, in order 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 7-4.
- 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 "The Generate RowIterator Class button" on page 7-3 for more information.

To close the Column designer, double-click any UI component, such as `contentPane`, in the content pane, or click a different component, and select Activate Designer. 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 `isVariant.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.

To format and edit 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

Creating a user interface using dbSwing components

Database application development is a feature of JBuilder Professional and Enterprise.

Distributed application development is a feature of JBuilder Enterprise.

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 DataStore-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 tutorials” on page 13-2 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 Select the Frame file. Click the Design tab.
- 2 Select the component on the component palette.
- 3 Click in the UI designer to place the component in the application.
- 4 Select the component in the component tree or the UI designer (The designer displays black sizing nibs on the edges of a selected component.)
- 5 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 provides a few of the dbSwing components available from the dbSwing tab of the component palette:

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

dbSwing offers significant advantages over Swing with increased functionality and data-aware capabilities. If you are considering building new applications, especially those that access databases, dbSwing is probably your best choice.

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.

Tutorial: Using dbSwing components to create a database application UI

Before you create a UI, you need to create an application, connect to a database and retrieve data from a data source.

See Chapter 4, “Connecting to a database” and “Querying a database” on page 5-14 for more information on how to do this, or follow the “Retrieving data for the tutorials” on page 13-2 for an example.

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, and is covered in the chapter “Creating a user interface” in *Building applications with JBuilder*.

You have two choices on how to proceed to create user interface with the JBuilder UI designer:

- Use a standard Java layout manager and let it control the placement and size of the components using the `constraints` property values you assign. This requires familiarity with the behavior of each of the layout managers, and can be confusing and frustrating to a new Java developer.

- Use **null** layout or `XYLayout` to prototype your UI, then convert to the final layouts after the visual components are in place. The advantage to this is that with these two layouts, you can size and align the components as you design. JBuilder will then use the size and location of the components in the UI designer to set their constraints when you convert to a different layout.

To learn about using layouts, see the online help topics “Laying out your UI”, and “Using layout managers” in *Building Applications with JBuilder*.

The steps below add the following UI components to a `BorderLayout` panel 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 Click the Design tab on `Frame1.java` to open the UI designer, then click on `contentPane` (`BorderLayout`) in the component tree to select it. The UI designer displays black square sizing nibs around the selected component’s outer edges in the UI designer.
- 2 Click the dbSwing tab on the component palette, then click the `JdbNavToolBar`.
- 3 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. `jdbNavToolBar1` automatically attaches itself to whichever `StorageDataSet` has focus.

`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.

- 4 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.
- 5 Add a `JdbStatusLabel` component, using the same method. Drop it in the area near the center, bottom edge of the panel. `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.
- 6 Add a `TableScrollPane` component to the center of the panel. Make sure its `constraints` property is `CENTER`. A table should fill the rest of the panel.

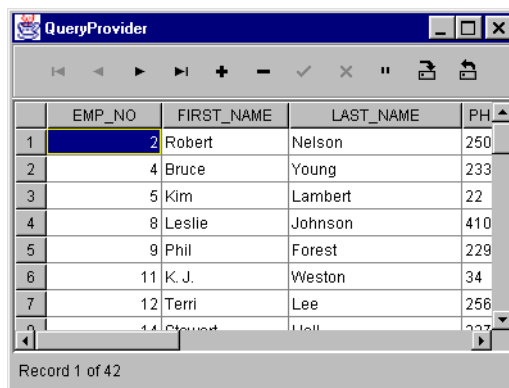
Scrolling behavior is not available by default in any Swing component or dbSwing extension, so, to get scrolling behavior, you must add scrollable Swing or dbSwing components to a `JScrollPane` or a `TableScrollPane`. `TableScrollPane` provides special capabilities to `JdbTable` over `JScrollPane`. See the dbSwing documentation for more information.

- 7 Finally, drop a `JdbTable` into the middle of the `tableScrollPane1` component in the designer. This fills the `tableScrollPane1` container with `jdbTable1`. In the Inspector, set the `dataSet` property for `jdbTable1` to whichever data set you want to view, for example, `queryDataSet1`.

You'll notice that the designer displays a table that displays live data and column headers. `JdbTable` allows multi-line column headers. Enter a column caption like `"line1<NL>line2"` in the Inspector or `"line1\nline2"` in code to see both lines in the table's header. Also, you can sort data by clicking on a table header in the running application.

- 8 Select `Run | Run Project` to run the application and browse and edit the data set.

The running application will look something like this:



To save changes back to the data source, you can use the Save button on the toolbar tool or, for more control on how changes will be saved, create a custom data resolver. See Chapter 8, “Saving changes back to your data source” for information on using other data resolving mechanisms.

The DataExpress samples (the dx subdirectory of samples) use dbSwing components. Other samples are located in the dbswing samples directory of your JBuilder installation.

Blocking editing in JdbTable

JdbTable’s `editable` property blocks cell editing as well as insertion and deletion of rows through its popup menu. However, it does not block a `JdbNavToolBar` from inserting or deleting rows. This is because `JdbNavToolBar` does not insert and delete rows in a `JdbTable`; it inserts and deletes rows into a data set. The `JdbTable` is updated to reflect those actions. To prevent these changes, hide or disable the toolbar’s Insert and Delete buttons or set the data set’s `enableInsert` and `enableDelete` properties to `false`.

Using other controls and events

Database application development is a feature of JBuilder Professional and Enterprise. Distributed application development is a feature of JBuilder Enterprise.

This topic provides more information on using controls and events. “Retrieving data for the tutorials” on page 13-2 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
- Displaying status information
- Accessing data and model information from a UI component
- Handling errors and exceptions

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 serves both as general step-by-step instructions for your real-world application, and as a tutorial with sample code and data.

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 topic "Retrieving data for the tutorials" on page 13-2.

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 `AppBrowser` to open it in the content pane, then click the `Design` tab that appears at the bottom of the `AppBrowser`.
- 3 Click the `dbSwing` tab 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. Select the `Last_Name` and `First_Name` columns and set the `required` property to `true` for both in the `Inspector`. Set the `SALARY` column's `min` property to 25000.
- 6 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 is presented with the message of the error. This Exception dialog has a Details button that displays the stack trace.
- If the `DataSetException` has chained exceptions, they can be viewed in the Exception dialog using the Previous and Next buttons.
- If the exception thrown is `ValidationException` (a subclass of `DataSetException`), the Exception dialog 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.

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 `ExceptionHandler` 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.

Creating a distributed database application using DataSetData

Database application development is a feature of JBuilder Professional and Enterprise.

Distributed application development is a feature of JBuilder Enterprise.

The `DataSetData.jpr` 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`'s Save button.

For more information on developing distributed applications, see Part III, "Distributed Application Developer's Guide" in the *Enterprise Application Developer's Guide*.

For more information on writing custom providers, see "Writing a custom data provider" on page 6-12. For information on writing or customizing a resolver, see "Customizing the default resolver logic" on page 8-16.

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.jpr`, 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 be 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

- 1 Open this application in JBuilder by selecting File | Open and browsing to /samples/DataExpress/StreamableDataSets/DataSetData.jpr.
- 2 Select Project | Project Properties to view the properties of this project. Set the following options;
 - Select the Run tab.
 - Check that the “java.rmi.server.codebase” property passed to the RMI server’s VM via a command-line argument points to the proper location of the RMI server’s classes (“file:/usr/local/jbuilder/samples/DataExpress/StreamableDataSets/classes/” by default).
 - Check that the “java.security.policy” property points to the SampleRMI.policy file included with this project (“file:/usr/local/jbuilder/samples/DataExpress/StreamableDataSets/SampleRMI.policy” by default).
 - Close the Project Properties dialog.
- 3 Start the RMI registry by selecting Tools | RMI Registry from JBuilder. The registry is toggled on and off from the Tools menu.
- 4 Select the file `DataServerApp` in the project pane. Right-click the file, and select Run to start the RMI server.
- 5 Select the file `ClientApp` in the project pane. Right-click the file, and select Run to start the RMI client.
- 6 Edit the data in the `ClientApp`’s table, and press 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?

These steps enable the `DataServerApp` to register itself as an RMI server. `DataServerApp` will respond to two RMI client requests: `provideEmployeeData` and `resolveEmployeeChanges`, as defined in the RMI remote interface `EmployeeApi.java`.

The `ClientApp` file is a frame with a `JdbTable` and a `JdbNavToolBar` for displaying data in a `DataExpress DataSet`. Data is provided to the `DataSet` 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. `DataServerApp` subsequently queries 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 resolve 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 accessible 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 3-tiers

To deploy the application on 3-tiers,

- Select `DataServerApp.java` in the project pane. 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.

- Select Project | Make Project to recompile and update `DataServerApp.class`.
- Deploy `DataServerApp.class` to a remote machine to which you are connected. `DataServerApp` runs on the middle, or second, tier.
- Start the RMI Registry on the middle tier computer.
- 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.

- Double-click `ClientFrame.java` in the project pane of JBuilder to bring it into 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.
- Select `clientResolver1` and modify the `hostName` property to the hostname of the middle-tier machine.
- Select 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 selecting Run.

For more information

- Read the RMI Documentation on the Sun web site at <http://www.java.sun.com/products/jdk/1.2/docs/guide/rmi/index.html>.
- Learn more about writing custom *Providers* and *Resolvers* by viewing the sample data set application `/samples/DataExpress/CustomProviderResolver/CustomProviderResolver.jpr`.
- Learn more about creating distributed applications in Part III, "Distributed Application Developer's Guide" in the *Enterprise Application Developer's Guide*.

Database administration tasks

Database application development is a feature of JBuilder Professional and Enterprise.

Distributed application development is a feature of JBuilder Enterprise.

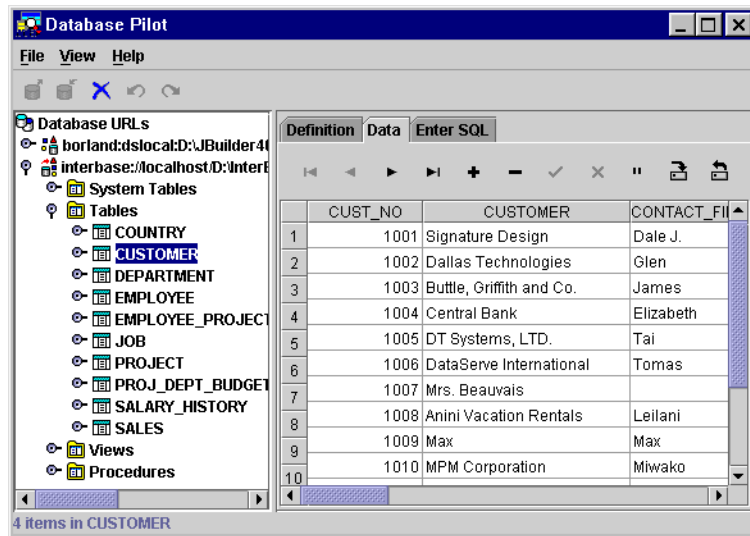
This chapter provides information on how to accomplish some common database administrator tasks. The following subjects are covered:

- Exploring database tables and metadata using the Database Pilot
- Using the Database Pilot for database administration tasks
- Monitoring database connections

Exploring database tables and metadata using the Database Pilot

The Database Pilot is a hierarchical database browser that also allows you to edit data. It 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.

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



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.
- Create, view, and edit data in existing tables.
- Enter and execute SQL statements to query a database.

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 selecting Tools | Database Pilot from the menu, then refer to its online book, *Database Pilot*.

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>/jdatastore/jdbcExplorer.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, and modify database URLs. The following steps assume the URL is closed, and lists each task, briefly describing the steps needed to accomplish it:

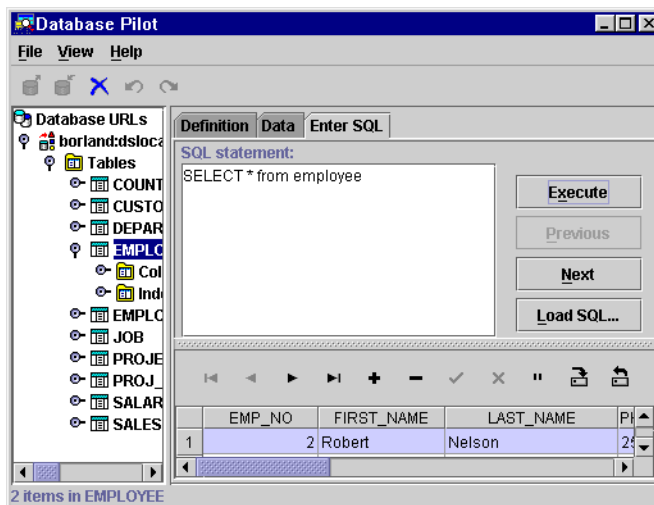
- View an URL
 - 1 In the left pane, select the URL to view. The Definition page appears in the right pane.
 - 2 Click the expand icon beside a database URL (or double-click it) in the left pane to see its contents.
- Create an URL
 - 1 Select an URL or database in the left pane.
 - 2 Right-click to invoke the context menu.
 - 3 Choose New (or select File | New from the menu).
 - 4 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.
 - 5 Browse to or enter the desired URL.

- 6 On the Definitions page in the right pane, specify the UserName and any other desired properties.
 - 7 Click the Apply button on the toolbar to apply the connection parameters.
- Modify an URL
 - 1 Select the URL to modify in the left pane. The Definitions page appears in the right pane.
 - 2 Edit settings on the Definitions page as desired.
 - 3 Click the Apply button on the toolbar to update the connection parameters.
 - Delete an URL
 - 1 Select the URL to delete in the left pane.
 - 2 Select File | Delete from the menu 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 three buttons, the Execute button, the Next button, and the Previous button. When an 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:



To query a database using SQL:

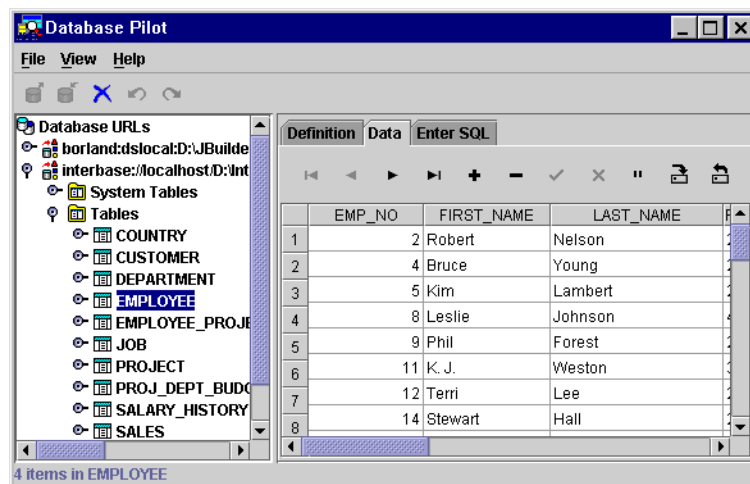
- 1 Open a database by selecting its URL in the left pane and entering user name and password if applicable.
- 2 Select the database or one of its child nodes in the left pane.
- 3 Click the Enter SQL tab in the right pane to display an edit box where you can enter or select an SQL statement.
- 4 Enter (or paste) an 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 Live Queries box of the Query page of the View | Options 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 list of tasks briefly describes the steps needed to accomplish each.

- View table data
 - 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
 - 1 Make sure that Request Live Queries in the View | Options menu 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 toolbar's Post button.
 - 4 To cancel an edit before moving to another record, click the toolbar's Cancel button or press *Esc*.
 - 5 To save your changes to the database, click the Save changes button.
- Insert a new record
 - 1 Place the cursor on the row before which you wish to insert another row.
 - 2 Click the toolbar's Insert button. 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 toolbar's Post button.
 - 5 To cancel an insert before moving to another record, click the toolbar's Cancel button or press *Esc*.
 - 6 To save an insert to the database, click the Save changes button.
- Delete a record
 - 1 Place the cursor on the row you wish to delete.
 - 2 Click the toolbar's Delete button.

Edits only take effect when they are applied. To apply edits and make changes permanent:

- Click the Post button on the toolbar. This posts the changes to the local data set only (not the database).
- Click the Save changes button to commit the edits to the database.

Using the Database Pilot for database administration tasks

This section provides an introduction to creating, populating, and deleting tables in an 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 Select Tools | Database Pilot.
- 2 From the Database Pilot, select File | New, or right-click an existing URL and select New from the context menu. The New URL dialog displays.
- 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.
- 6 On the Definitions page in the right pane, specify the UserName and any other desired properties.
- 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 Select **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 an 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 Live 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 18-7.
- 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 You can now use the Database Pilot to view, edit, insert, and delete data in tables. See "Using the Explorer to view and edit table data" on page 18-5 for more information on these tasks.

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 18-7 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.

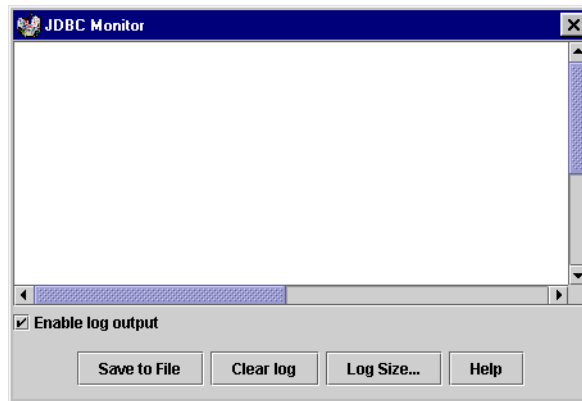
Monitoring database connections

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 18-10.

JDBC Monitor will monitor any JDBC driver (i.e., any subclass of `java.sql.Driver`) while it is in use by JBuilder. The JDBC Monitor monitors all output directly from the JDBC driver.

Understanding the JDBC Monitor user interface

To start the JDBC Monitor, select Tools | JDBC Monitor. The JDBC Monitor displays:



How to use 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.

Pressing 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 Select Tools | Configure Palette.
- 2 Select Data Express from the Pages field on the Pages tab.
- 3 Select the Add Components tab.
- 4 Select JBCL 3.1 in the Select Library field.
- 5 Click Install.
- 6 Browse to `com.borland.jbcl.sql.monitor.MonitorButton`.
- 7 Click OK to close the dialog.

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.

Sample database application

Database application development is a feature of JBuilder Professional and Enterprise. Distributed application development is a feature of JBuilder Enterprise.

This chapter describes a sample database application developed using DataExpress components and the JBuilder design tools. Where necessary, the code generated by the design tools was modified to provide custom behavior. There are no tutorial steps on this application as it is intended to consolidate the individual how to topics discussed elsewhere in this book.

The completed files for this sample application are included in the `/samples/DataExpress/BasicApp` subdirectory of your JBuilder installation under the project name `BasicApp.jpr`. The file `BasicApp.html` contains updated information on this sample application. If you experience problems running this application, see “JBuilder sample files” on page 3-2 for information critical to this process. (If you downloaded JBuilder, you also need to download the Samples Pack in order to have this sample.)

Note Some of the samples run only with JBuilder Enterprise.

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 5-14.)
- Contains a `JdbTable` which displays the 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 7-7 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 14-15.)
- Data editing that is controlled using edit masks (the `HIRE_DATE` column). (See “Adding an Edit or Display Pattern for data formatting” on page 14-15.)
- 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 14-7.)
- 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 16-2.)
- 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 13-14.
- 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 “Saving changes from a `QueryDataSet`” on page 8-2.) The Save button of the `JdbNavToolBar` performs the save. Messages regarding the resolve process are displayed in the `JdbStatusLabel` control.

Sample international database application

Writing an international application involves additional complexities which impact application development, for example, using locales other than `en_US` (American English). For an example of an international database application that includes many common features of a database application in addition to support for multiple languages and locales, see the `/samples/dbSwing/MultiLingual` directory of your JBuilder installation. (If you downloaded your JBuilder, you also need to download the Samples Pack in order to have this sample.)



Database FAQ

The database FAQ, a document comprised of answers to questions posted on the JBuilder Database newsgroup, is posted on the Borland web site. The most current version of this document is available on the Borland Community Web site at <http://community.borland.com/>, as of this printing.

This document may also be posted on the newsgroup and updated there periodically. If you do not find what you are looking for in this document, post a question to the newsgroup `borland.public.jbuilder.database`. To access the Database newsgroup, point your browser to <http://www.borland.com/newsgroups/#jbuilder>, and select `borland.public.jbuilder.database`.

A dbSwing FAQ is located on the dbSwing newsgroup. If you do not find what you are looking for in this document, post a question to the newsgroup `borland.public.jbuilder.dbswing`. To access the dbSwing newsgroup, point your browser to <http://www.borland.com/newsgroups/#jbuilder>, and select `borland.public.jbuilder.dbswing`.

A JDataStore FAQ is located on the Borland Community Web site at <http://community.borland.com/>. If you do not find what you are looking for in this document, post a question to the newsgroup `borland.public.jbuilder.jdatastore`. To access the JDataStore newsgroup, point your browser to <http://www.borland.com/newsgroups/#jbuilder>, and select `borland.public.jbuilder.jdatastore`.

Borland provides newsgroups as a user-supported area in which to exchange information, tips, and techniques on our products for the global community of Borland and Inprise customers.

We encourage members of this community to assist each other with technical questions. We have established a special relationship with a group of volunteers known as TeamB. TeamB members come from a wide

range of backgrounds and professions, with a willingness to give their time, expertise, and advice to enhance the technical skills of other customers. This forum is often visited by JBuilder R&D and QA engineers, as well as Borland Developer Support, however, Borland does not offer any formal support in these newsgroups, except for questions on product installation.

Index

Symbols

? as JDBC parameter marker 5-33

A

accessing data 4-1, 5-1
 from custom data sources 6-12
 from data module 11-1
 from UI components 16-2
 JDBC data sources 4-1
accessing model information 16-2
adding columns
 for internal purposes 7-9
 to imported text files 10-4
 to parameterized queries 5-33
adding components
 to data modules 11-3
adding parameters to queries 5-27
agg property editor 14-14
AggDescriptor objects 14-14
aggregating data 14-1
 creating aggregated columns 14-10, 14-14
 customizing aggregation methods 14-15
 tutorial
 aggregated columns 14-11
Aggregating sample 14-11
Application Generator 11-18
application structure
 developing 5-5
Application wizard
 in tutorial 5-5
applications
 client-server 11-8
 database (2-tier) 11-8
 developing
 text file 5-5
 generating 11-18
Apply button 11-13
ascending sort order 11-14
ASCII files *See* text files

B

BasicApp sample 19-1
binding parameter values 5-34
boolean data
 patterns 14-21
boolean patterns 14-15
 examples of 14-18
bundling resources 5-22
business logic 11-1, 11-4

C

caching data 12-1
calculated columns 14-1, 14-7, 14-10
 aggregating data in 14-14
 aggregation 14-10, 14-11
 creating lookups with 14-2, 14-5
 creating picklists with 14-3
 tutorial 14-8
 types supported 14-7
CalculatedColumn sample 14-8
cascadeDeletes 9-3
cascadeUpdates 9-3
client-server applications 11-8
 creating 11-8
closing data sets 5-26
coding events
 for data modules 11-4
colors 7-1
Column component 5-4, 14-26
 formatter property
 using 14-26
 locale property 10-8
 manipulating 14-24
 overview 2-8, 7-1
 persistent 14-23, 14-24
 setting properties 14-23, 14-24
 specifying as persistent 7-7
 storing Java objects in 14-26
 using 5-4
 viewing 14-23
Column designer 7-2, 14-23, 14-24
 enabling 7-2
 metadata options 7-4
 RowIterator Generator button 7-3
column order
 locating data 13-19
Column properties
 for multiple table queries 8-13
column properties 5-4
columns 5-4, 7-1, 7-7
 adding to StorageDataSets 7-9
 calculated 14-1
 changing properties for 7-2
 controlling order of 7-10
 definitions, in SCHEMA files 10-6
 exploring 18-1
 filtering data in 13-1
 linking on common 9-2
 lookup values in *See* lookup columns
 metadata 5-4

- persistent columns 5-4
- properties 5-4
- setting persistent properties for 7-6
- setting properties for 7-1, 7-2
- sorting 13-1
- viewing information 7-2, 7-5
- working with 7-1
- com.borland.datastore package 2-6
- com.borland.dx.dataset package 2-6
- com.borland.dx.sql.dataset package 2-6
- comma-delimited text files 10-2
 - importing 10-2
- common fields 9-1
- components
 - DataExpress 5-2
 - JFC data-aware 16-1
 - synchronizing 16-1
- .config files
 - creating for drivers 3-7
- connecting to DataExpress component 5-12
- connection pooling 4-10
- connection problems
 - solutions 4-9
- connections 3-2, 4-1
 - overview 4-1
 - troubleshooting 3-9
 - tutorial 4-2
- constraints
 - enabling 13-12
- controlling user input 14-17
- controls 16-1
- cost of goods 14-15
- Create ResourceBundle dialog 5-22
- creating master-detail relationships 9-1, 9-5
- creating queries 5-14
 - tutorial 5-16
- creating SQL tables 18-7
- cursors
 - shared 16-1
- custom providers 6-12
- custom resolving 8-17

D

- data 5-1
 - accessing 13-2
 - alternate view of 14-22
 - caching 12-1
 - editing 18-5
 - exploring 18-9
 - exporting 10-5
 - from a QueryDataSet 10-10
 - extracting from a data source 5-1
 - filtering 13-1
 - finding 13-14
 - importing 5-4

- inserting 18-9
- loading 6-14
- locating 13-1, 13-14, 13-16, 13-17
- modifying 18-9
- persistent 14-23
- persisting 12-1
- providers 5-1
- providing 5-1, 6-14
- relationships
 - 1-to-1 8-12
- required 14-24
- resolving 8-1, 8-2
 - customizing 8-20
 - stored procedures 8-5
- resolving data
 - default behavior 8-17
- retrieving 5-1, 5-16, 6-12, 13-2
- sorting 13-1
- storing 12-1
- viewing 18-5
- data constraints
 - enabling 13-12
- data fields
 - exporting 10-8
- data filters 13-5
 - tutorial 13-6
- data groups 14-10
- data input
 - patterns 14-1
- data members
 - nontransient 8-15
 - private 8-15
- Data Modeler
 - 2-tier applications 11-8
 - client-server applications 11-8
 - creating queries 18-2
- data models 11-1
- data modules
 - adding business logic 11-4
 - adding components 11-3
 - adding to libraries 11-5
 - class files 11-5
 - compiling 11-5
 - creating 11-2, 11-8
 - referencing 11-5, 11-7
 - saving 11-5
 - using 11-5, 11-7
 - using generated 11-19
 - wizards 11-2, 11-7
- Data page
 - Database Pilot 18-5
- data patterns 14-15
 - examples of 14-18
- data providers 5-1
- data retrieval enhancements 5-24

- data sets
 - binding parameter values 5-34
 - closing 5-26
 - enhancing performance 5-24
 - explicitly opening 5-26
 - linking 9-1
 - opening 6-14
 - returning as read-only 5-26
 - streamable 8-14
- data sources
 - accessing 5-1
 - connecting to 3-2
 - updating 8-16
- data streams
 - storing 12-1
- data summaries 14-10, 14-11
- data tables
 - displaying detail link columns 9-4
- data types
 - variant 14-26
- data-aware components 16-1
 - default data display 7-2
 - displaying columns in 14-23
- database
 - functionality
 - setting up 3-1
- database administration 18-1, 18-7
- database applications 2-1
 - creating 13-2
 - distributed 17-1
 - errors 3-9
 - generating 11-18
 - introduction 2-1
- Database component 4-1, 4-3
 - overview 4-1
 - tutorial 4-3
 - using 4-3
- Database components
 - DataExpress
 - architecture 2-1
 - overview 2-6
- database connections
 - monitoring 18-9, 18-10
 - pooling 4-10
- database drivers
 - adding to JBuilder 3-7
 - adding to project 3-8
 - all-Java 3-4
 - setting up 18-3
- Database field
 - in QueryDescriptor 5-20
- database newsgroups A-1
- Database Pilot 7-5
 - Data page 18-5, 18-9
 - database drivers
 - setting up 18-3
 - Enter SQL page 18-4
 - using 18-1
 - viewing column information 7-5
 - window 18-2
- database sample files
 - installing 3-2
- database samples
 - for Unix users 3-3
- database servers
 - communicating with 2-1
- database tutorials 8-16, 19-1
 - adding status information 16-3
 - alternate views 14-22
 - calculated aggregations 14-11
 - calculated columns 14-8
 - creating lookups 14-5
 - creating picklists 14-3
 - creating queries 5-16
 - creating stored procedures 6-3, 6-8
 - exporting data 10-6
 - filtering data 13-6
 - importing comma-delimited data 10-2
 - importing formatted data 10-4
 - international application 19-2
 - JDBC connections 4-2
 - locating data 13-14
 - master-detail relationships 9-6
 - parameterizing queries 5-27
 - QueryDataSet components
 - resolving changes 8-2
 - reading from text files 5-4
 - ResolverEvents 8-19
 - resolving ProcedureDataSets 8-5
 - setting up 3-1
 - setting up JDataStore 3-4
 - setting up under Unix 3-3
 - sorting data 13-10
 - StreamableDataSets 17-3
 - using dbSwing components 15-2
 - viewing column information 7-2
- database-related packages 2-6
- databases 4-3
 - connecting to 3-2, 4-3
 - connecting via JDBC 8-2
 - connections
 - troubleshooting 3-9
 - deploying 3-8
 - exploring 18-1
 - in distributed applications 17-1
 - indexes 18-1
 - information
 - displaying 18-2
 - local
 - accessing 18-3

- properties (tutorials) 13-1, 14-1
- querying 5-19
- remote
 - accessing 18-3
- technical assistance with A-1
- troubleshooting A-1
- UI 13-1, 14-1
- DataExpress
 - applications 2-1
 - architecture 2-1, 2-4
 - DataStore 12-1
 - components 2-4, 2-6, 5-1
 - accessing data with 5-1
 - tutorial for adding 5-7, 5-8
 - connecting to UI component 5-12
 - overview 5-2
 - tutorials 5-2
- DataExpress Component Library
 - described 2-1
- DataModule interface
 - customizing 11-3
 - discussed 11-1
 - overview 2-9
 - referencing 11-7
- DataRow class
 - locating data 13-17
- DataRow component
 - column order in locates 13-19
 - overview 2-8
 - using to locate data 13-16
- DataSet
 - caching in DataStore 12-1
 - filtering data in 13-1
 - functionality 2-4
 - locating data in 13-1
 - persisting in DataStore 12-1
 - saving changes 8-1
 - sorting data in 13-1
 - storing Java objects 14-26
 - streamable
 - serializable 8-14
 - with RMI 8-14
- DataSet class
 - overview 2-6
 - using 5-3
- DataSet package 2-6
- DataSetData
 - extractDataSet method 8-15
 - using 8-14
 - extractDataSetChanges method 8-15
 - metadata
 - passing 17-4
 - populating 8-15
 - sample application, described 17-2
 - tutorial 17-1
- DataSetData objects 8-14
- DataSetData sample application 17-1
- DataSetData.jpr
 - running 17-3
- DataSetException 16-4
- DataSetView component
 - overview 2-8
 - sorting in 13-9
 - using 14-1
- DataStore
 - creating 12-3
 - data streams 12-1
 - database files
 - accessing 5-3
 - operations 12-3
 - overview 2-7
 - using 12-1
 - verifying 12-3
 - when to use 12-1
- DataStore Explorer 12-2
- DataStore package 2-6
- DataStoreDriver component
 - overview 2-7
- date data
 - patterns 14-19
- dates
 - importing 10-4
- DBA 18-7
 - tasks 18-1
- dbSwing
 - creating database UI using 15-1
 - FAQ A-1
- dbSwing components
 - creating database UI 15-1
 - sample applications using 15-2
 - tutorial 15-2
 - using 16-1
- dbSwing sample applications
 - MultiLingual 19-2
- Default project
 - adding database drivers 3-8
- defaults 7-2
 - data display 7-2
- Delay Fetch of Detail Records Until Needed 9-3
- delete procedures
 - custom 8-9
- deleteProcedure property 8-8
- deleting persistent columns 7-8
- deleting tables 18-9
- delimiters 5-2
- deployment
 - of JDBC drivers 3-8
- descending sort order 11-14
- designing applications 5-9

- detail records
 - fetching 9-3
- detail tables 9-2
 - editing 9-4
- developing projects 5-5
- development FAQs A-1
- discounts 14-15
- display masks 14-1, 14-17
 - adding 14-15
- displaying data
 - in data-aware components 7-2
- displaying special characters 14-17
- displaying status information 16-2
- distributed database applications 17-1
- distributed objects
 - database 17-1
- documentation conventions 1-5
- driver manager 4-1
- drivers
 - adding database drivers to project 3-8
 - adding JDBC driver 3-7
- drivers, databases, setting up 18-3

E

- edit masks 14-1
 - adding 14-15
 - editMask property 14-17
- edit/display masks 14-1
- editing
 - blocking 15-5
- editing data
 - controlling user input 14-17
 - master-detail 9-4
- enableDelete property
 - of DataSetView 14-22
- enableInsert property
 - of DataSetView 14-22
- enableUpdate property
 - of DataSetView 14-22
- Enter SQL page
 - Database Pilot 18-4
- errors
 - handling 16-4
 - in database 3-9
- escape sequences 6-7
- event handlers
 - custom aggregation 14-15
- events
 - adding business logic 11-4
 - resolver 8-16
- examples
 - stored procedures 6-10, 6-11
- exceptions 16-4
 - handling 16-4

- Execute Query command 11-15
- export masks 14-15, 14-17
- exportDisplayMask 14-17
 - property
 - example 10-8
- exporting data 10-5
 - from a QueryDataSet 10-10
 - to text files 10-1
 - using patterns 10-8
- extractDataSet method 8-15
- extractDataSetChanges method 8-15
- extracting data 5-1

F

- FAQ A-1
 - database A-1
 - dbSwing A-1
 - JDataStore A-1
- fetchAsNeeded 9-3
- fetching data 5-14
 - detail records 5-35, 9-3
 - from JDBC data sources 10-5
 - from JDBC sources 6-1
 - optimizing 5-24
- fetching detail records 9-3
- field separators 5-2
- fields
 - databases
 - exploring 18-1
 - linking on common 9-2
 - required 14-24
- filtering data 13-1, 13-5
 - tutorial 13-6
- FilterRows sample 13-6
- flat file databases 10-1
- formatted text files 10-4
 - importing 10-4
- formatter property
 - using 14-26
- formatting data 14-15
 - display masks for 14-17

G

- generating applications 11-18
- Group By clause 11-12
- Group By page
 - Data Modeler 11-12
- grouping data 14-10

H

- handling errors 16-4
- handling exceptions 16-4

I

- import masks 14-15, 14-17
- importing data 5-3, 12-3
 - file-based data 5-3
 - from text files 10-1
 - TextDataFile sample 5-4
 - tutorial 5-4
 - tutorials for 10-2, 10-4
- indexes
 - database 18-1
 - unique vs. named 13-12
- insert procedures
 - custom 8-9
- insertProcedure property 8-8
- insertRow() method 6-14
- installing
 - InterClient 3-4
 - JDataStore Server 3-4
 - sample files 3-2
- InterBase
 - about 3-4
 - setting up for JBuilder 3-4
 - tips 3-6
- InterBase stored procedures
 - example 6-10
 - return parameters 8-11
- InterClient
 - about 3-4
 - connection errors 4-9
 - installing 3-4
 - setting up for JBuilder 3-4
 - setting up in JBuilder 3-7
 - using JDBC drivers 4-6
- INTERNALROW 8-14, 8-15
- Internet
 - developing client-server applications 3-4
- InterServer 3-4
- Intranet
 - developing client-server applications 3-4

J

- Java
 - data 3-4
 - objects containing DataSets 8-14
 - RMI with databases 17-1
 - server applications
 - responding to client requests 8-14
- Java data modules
 - saving queries 11-17
- Java interfaces
 - developing with InterClient 3-4
- JBCL components
 - data-aware 16-1
- jbInit() method 11-19

- JBuilder
 - database functionality 3-1
- JDataStore
 - FAQ A-1
 - installing local server 3-4
 - when to use 5-2
- JDataStore JDBC drivers 3-4
- JDataStore Server
 - installing 3-4
- JDBC 8-2
 - connection errors 3-9
 - pooling connections 4-10
- JDBC API 2-1, 4-1
- JDBC connections 4-1
 - managing 2-6
 - manipulating traffic 18-9
 - monitoring 18-9
 - overview 4-1
 - tutorial 4-2
- JDBC data sources 5-1, 6-1, 10-5
 - accessing 4-1, 5-1
 - from text files 10-11
 - saving text file data 10-11
- JDBC driver
 - adding to JBuilder 3-7
 - adding to project 3-8
- JDBC drivers 3-1, 3-2
 - adding to JBuilder 3-7
 - InterClient 4-6
 - JDataStore JDBC drivers 3-4
 - setting up 3-4
 - when to use 5-2
- JDBC escape sequences 6-7
- JDBC Monitor 18-9
 - in applications 18-10
 - using 18-9
- JdbNavField component 13-14
 - example 13-14
- JdbStatusLabel component 16-3
 - tutorial 16-3
- JdbTable
 - blocking editing 15-5
- JdbTable component
 - sorting in 13-9
- JFC components 16-1
- joining tables 9-1

L

- libraries
 - adding to project 11-5
 - creating 3-7
 - required 11-5
- Link Queries dialog box 11-16
- linked tables
 - types 8-12

- linking data sets 9-1
- load opt 5-20
- Load Options field
 - in QueryDescriptor 5-20
- loading data 6-14
- Local InterBase Server 3-6
- locale property 10-8
- locale-specific resources 5-22
- locate method 13-16
- LocateOptions class 13-17
- locating data 13-1, 13-14, 13-17
 - column order 13-19
 - interactive 13-14
 - locate options 13-17
 - programmatically 13-16
 - variants 13-19
- lookup columns 14-2
 - creating 14-2
 - tutorial 14-5
- lookup lists 14-1

M

- manipulating JDBC traffic 18-9
- many-to-many data relationships 8-12
- many-to-one data relationships 8-12
- masks 14-1
 - for data formats 14-17
 - for editing 14-17
 - for importing/exporting 14-17
- master tables 9-2
 - editing 9-4
- masterDataSet 9-5
- master-detail relationships 11-16
 - creating 9-1, 9-5
 - defining 9-2
 - queries 5-35
 - resolving 9-10
 - custom 8-21
 - tutorial 9-6
- MasterDetail sample 9-6
- masterLink 9-2
- MasterLinkDescriptor class
 - usage overview 9-2
- metadata 7-1
 - discovery 7-1
 - exploring 18-1
 - obtaining 6-13
 - persisting 5-25, 7-4
 - setting as dynamic 7-4
 - updating in persistent columns 7-8
 - viewing 7-5
- metaDataUpdate property
 - with multiple tables 8-13
- middle-tier server implementations 8-14

- models
 - accessing information about 16-2
- MonitorButton
 - adding to palette 18-11
 - properties 18-11
 - using 18-11
- monitoring connections 18-9, 18-10
- monitoring JDBC drivers 18-9
- multi-column locates
 - column order 13-19
- MultiLingual sample application 19-2
- multi-table resolution 8-11
 - resolution order 8-14

N

- named indexes 13-12
- named parameters 5-33
- navigating
 - multiple data sets 16-1
 - synchronizing components 16-1
- New Data Module Wizard 11-2
- newsgroups
 - database support A-1
- nontransient data members 8-15
- non-visual components 5-7
 - setting properties 5-8
- numeric data 10-4
 - importing 10-4
 - patterns 14-18
- numeric fields
 - exporting 10-8
- numeric patterns 14-15
 - examples of 14-18

O

- objects
 - containing DataSets 8-14
 - Java 14-26
 - storing 14-26
- one-to-many relationships 8-12, 9-1
- one-to-one relationships 8-12
- opening data sets 5-26
- optimizing data retrieval 5-24
- Oracle PL/SQL stored procedures
 - example 6-10
- Order By clause
 - adding 11-14
- Order By page 11-14

P

- packages
 - database-related 2-6
- parameter markers 5-33

- parameterized queries 5-27, 11-13, 11-15
 - adding columns 5-33
 - binding values 5-34
 - for master-detail records 5-35
 - supplying new values 5-35
 - tutorial 5-27
- ParameterRow 5-28
- ParameterRow component 5-33
 - overview 2-9
- parameters
 - return 8-11
 - specifying 5-21
- Parameters tab
 - QueryDescriptor 5-21
- parsing data 14-15
- parsing strings 14-17
- PARTIAL option
 - multi-column locates 13-19
- password
 - prompting for 4-9
- Paste Column button 11-13
- Paste Parameter button 11-13
- patterns 14-1, 14-15
 - boolean data 14-21
 - date data 14-19
 - examples of 14-18
 - for data entry 14-17
 - for exporting data 10-8
 - numeric data 14-18
 - string data 14-20
 - time data 14-19
- performance 5-24
- Persist all Metadata option 7-4
- persistence 12-1
- persistent columns 14-24
 - adding 7-9
 - controlling metadata update with 7-8
 - deleting 7-8
 - overview 7-7
- persistent data 14-23
- persisting data 12-1
- picklists 14-1, 14-2
 - removing 14-4
 - tutorial 14-3
- Place SQL Text In Resource Bundle
 - in QueryDescriptor 5-20, 5-22
- populating SQL tables 18-9
- private data members 8-15
- procedure calls 5-3
 - server-specific 6-7
- ProcedureDataSet
 - resolving data 8-6
 - saving changes to 8-9
- ProcedureDataSet component 5-3
 - about 6-1
 - overview 2-7
 - sorting in 13-9
 - tutorial 6-3, 6-8
 - using 5-3
- ProcedureDataSet components
 - DataExpress
 - architecture 2-1
 - tutorial
 - resolving 8-5
- ProcedureResolver 8-1
 - properties 8-8
 - using 8-5, 8-6
- ProcedureResolver component
 - coding 8-8
 - tutorial 8-9
- procedures 5-3
 - resolving 8-5
 - stored
 - running 5-3
- Project wizard
 - in tutorial 5-5
- projects
 - developing 5-5
- prompting
 - user name and password 4-9
- properties
 - non-visual components 5-8
- provideData method 6-14
- ProviderHelp
 - initData method 6-14
- providers 5-3
 - creating custom 6-12
 - custom 5-3, 17-1
 - of data 5-1
- providing (defined) 6-1
- providing data 5-3
 - for database examples 13-2
 - from JDBC sources 6-1
 - using custom provider 5-3
 - with parameterized queries 5-27

Q

- queries 5-3, 5-14
 - building 5-19
 - containing WHERE clause 8-12
 - creating 18-2
 - creating parameterized 5-27
 - creating with Data Modeler 11-8
 - editing directly 11-15
 - ensuring updateability 5-26
 - executing 18-4
 - Group By clause 11-12
 - master-detail 11-16

- multiple in Data Modeler 11-16
- on multiple tables 8-12
 - column properties 8-13
- optimizing 7-6
- overview 5-14
- parameterized 5-3, 11-13, 11-15
- required components 5-14
- saving to data module 11-17
- saving to Java data modules 11-17
- sort order 11-14
- SQL
 - Database Pilot 18-4
 - testing 11-15
 - tutorial 5-16, 13-2
 - viewing results 11-15
 - Where clause 11-13
- query property
 - parameters 5-21
 - understanding 5-20
- query property editor 5-19
- query statements 5-3
 - running 5-3
- Query tab 5-20
- QueryDataSet 5-29
 - saving changes 8-3
- QueryDataSet component 2-7
 - exporting to a file 10-5, 10-10
 - overview 5-14, 5-24
 - query property setting 5-19
 - sorting in 13-9
 - tutorial 5-16, 5-27
 - using 5-3
- QueryDataSet components
 - DataExpress
 - architecture 2-1
 - tutorial 8-2
- QueryDescriptor
 - Parameters tab 5-21
 - Place SQL text in resource bundle option 5-22
 - Query page 5-20
 - visual 5-19
- QueryProvider
 - for multiple table queries 8-13
- QueryProvider sample 5-16
- QueryResolver 8-1
 - saving changes with 8-6
 - with stored procedures 8-6
- QueryResolver component
 - adding 8-17
 - customizing 8-16, 8-17
 - events, controlling 8-17
 - intercepting events 8-17
- QueryResolver components
 - default 8-17

R

- read-only data sets 5-26
- reconciling data 8-1
- relational databases 9-1
- remote databases
 - connecting to 3-2
- remote servers 4-1
- removing persistent columns 7-8
- required data 14-24
- resolution order
 - specifying 8-14
- resolution process
 - controlling 8-17
- ResolutionManager class 8-16
- resolveOrder property 8-11, 8-14
- resolver events 8-16
- ResolverEvents sample application 8-19
- ResolverListener 8-17
- ResolverResponse 8-17
- resolvers
 - custom 8-1, 8-17, 8-20, 17-1
 - default 8-17
- resolving
 - tutorial 8-6, 8-8, 8-9
- resolving data 8-1, 8-2, 8-16, 8-20
 - customizing events 8-17
 - customizing resolver logic 8-17
 - default 8-17
 - master-detail relationships 9-10
 - multiple tables 8-11
 - QueryDataSets 8-3
 - stored procedures 8-5
- resource bundles 5-20, 5-22
- Resourceable SQL 5-22
- retrieving data 4-1, 5-1, 5-16, 6-12, 13-2
 - from a data module 11-1
 - from data sources 8-16
 - through stored procedures 6-1
- return parameters 8-11
- RMI
 - with databases 17-1
- RowFilterListener interface
 - tutorial 13-6
- rowID property
 - using 8-13
- RowIterator 7-3
 - using 7-3

S

- sales tax 14-15
- sample applications 19-1
 - database 3-2
 - DataSetData 17-2
 - international locales 19-2

- ResolverEvents 8-19
- samples
 - for Unix users 3-3
 - running 3-2
- saveChanges() method
 - and rowID property 8-13
- saving changes 8-2, 8-5, 8-16
 - master-detail relationships 9-10
 - to QueryDataSets 8-3
- saving data 8-1
 - multiple tables 8-11
 - tutorial 8-6, 8-8, 8-9
- SCHEMA files 10-4, 10-6
 - and exportDisplayMasks 14-17
- schemaName property 8-11
- Selected Sort Order Direction options 11-14
- serializing objects 8-14
- setResolver 8-17
- shared cursors 16-1
- SimpleStoredProcedure sample 6-3, 6-8, 8-6
- sort order 13-12
 - SQL queries 11-14
- sort property editor 13-10
- sorting data 13-1, 13-9
 - in tables 13-9
 - programmatically 13-13
 - sort order 13-12
 - with design tools 13-10
 - with master-detail relationships 9-2
- sortOnHeaderClick 13-9
- special characters 14-17
- specifying resolution order 8-14
- SQL Builder 5-19
- SQL connections 4-1
- SQL databases
 - connecting to 3-2
- SQL queries 5-14
 - adding parameters 5-27
 - editing directly 11-15
 - ensuring updateability 5-26
 - Group By clause 11-12
 - master-detail 11-16
 - multiple in Data Modeler 11-16
 - optimizing 7-6
 - overview 5-14
 - required components 5-14
 - resourceable 5-22
 - saving to data module 11-17
 - saving to Java data modules 11-17
 - sort order 11-14
 - testing 11-15
 - tutorial 5-16
 - view results 11-15
 - Where clause 11-13
- SQL server connections
 - troubleshooting 3-9
- SQL servers
 - connecting to *See* SQL connections
- SQL Statement field
 - in QueryDescriptor 5-20
- SQL statements
 - defining 5-19
 - discussion of 6-7
 - encapsulating 6-1
 - executing 18-4
- SQL tables 5-3
 - creating 18-7
 - deleting 18-9
 - from text files 10-10
 - populating 18-9
 - querying 5-3
 - saving text file data 10-10
 - updating 8-1
- SqlRes class 5-22
- SQLResolver 8-1, 8-16
 - customizing 8-17
 - using with multiple tables 8-11
- status information 16-2
- status label
 - adding to applications 16-2
- StatusEvent listener 16-4
- StorageDataSet class
 - overview 2-7
 - usage overview 5-1
- StorageDataSet component
 - adding empty columns 7-9
 - controlling column order 7-10
 - saving changes 8-1
- StorageDataSet methods
 - insertRow() 6-14
 - startLoading() 6-14
- storageDataSet property
 - in DataSetView 14-22
- store property 12-1
- stored procedures 5-3
 - creating 6-7
 - examples 6-10, 6-11
 - InterBase 8-11
 - overview 6-1
 - resolving 8-5
 - return parameters 8-11
 - running 5-3
 - tutorial 6-3, 6-8, 8-6, 8-8
 - ProcedureResolver 8-9
- streamable data sets 8-14
- streamable DataSets
 - using 8-14
- StreamableDataSets sample
 - running 17-3

- streaming data 8-14
- string conversions
 - with masks 14-17
- string data
 - patterns 14-20
- string patterns 14-15
 - examples of 14-18
- strings
 - parsing 14-17
- summarizing data 14-10, 14-11
- Sybase stored procedures
 - example 6-11
- synchronizing components 16-1
- synonyms
 - displaying data in 18-5

T

- table data
 - editing 18-9
 - viewing 18-9
- tableColumnName property 8-11
- TableDataSet component 10-8
 - overview 2-8
 - resolving 10-10
 - saving changes to 10-5
 - sorting in 13-9
 - tutorial
 - saving changes to 10-6
 - usage overview 10-1
- tableName property 8-11
- tables
 - creating 18-7
 - deleting 18-9
 - editing data 18-5
 - exploring 18-1
 - linked 8-12
 - not updateable 8-13
 - populating 18-9
 - querying 13-2
 - viewing data 18-5
- TestFrame.java sample 6-12
- testing queries 11-15
- text fields
 - exporting 10-8
- text files 10-1
 - applications 5-5
 - exporting 5-2, 10-1, 10-5
 - import tutorials 10-2, 10-4
 - importing 10-1, 12-3
 - text files
 - importing 5-2
 - to JDBC sources 10-11
 - to SQL tables 10-10
 - tutorial for extracting from 5-4

- TextDataFile component 5-2
 - resolving 10-11
 - retrieving JDBC data for 10-5
 - usage overview 10-1
 - using 5-2
- TextDataFile sample 5-4
 - compiling 5-12
- TextFileImportExport sample 10-2
- time data
 - patterns 14-19
- time fields
 - exporting 10-8
- time patterns 14-15
 - examples of 14-18
- totals 14-15
- transaction processing
 - default 8-1
- transactions 8-2
- troubleshooting
 - database connections 3-9
- tutorials 19-1
 - adding status information 16-3
 - calculated aggregations 14-11
 - calculated columns 14-8
 - creating lookups 14-5
 - creating picklists 14-3
 - creating stored procedures 6-3, 6-8
 - database 8-16, 13-14
 - databases
 - alternate views
 - DataSetView component
 - using 14-22
 - resolving data changes 8-2
 - using dbSwing components 15-2
 - DataExpress components 5-2
 - exporting data 10-6
 - filtering data 13-6
 - importing comma-delimited data 10-2
 - importing formatted data 10-4
 - installing 3-1
 - international application 19-2
 - JDBC connections 4-2
 - master-detail relationships 9-6
 - parameterizing queries 5-27
 - query (for database examples) 13-2
 - querying databases 5-16
 - reading from text files 5-4
 - ResolverEvents 8-19
 - sorting data 13-10
 - stored procedures 8-6
 - coding 8-8
 - ProcedureResolver 8-9
 - viewing column information 7-2
 - two-tier applications
 - generating 11-18

U

- UI components 5-12
 - tutorial for adding 5-9, 5-12
- UI designer
 - using 5-9
- UI elements
 - adding 5-9
- Unable to load dll 'JdbcOdbc.dll' error 3-9
- unique indexes 13-12
- updateProcedure property 8-8
- updating data
 - from data sources 8-16
- updating data sources 8-1
- URL
 - adding in Data Modeler 11-10
 - opening in Data Modeler 11-9
- Use Data Module Wizard dialog
 - explained 11-7
- user input
 - controlling 14-17
 - parsing 14-17
- user interfaces
 - tutorial for adding DataExpress components 5-7
 - tutorial for adding DataExpress components 5-8
 - tutorial for adding UI components 5-9, 5-12
- user name
 - prompting for 4-9

V

- ValidationException 16-4
- Variant class
 - locating data 13-19
- variant data types 14-26
- Variant.OBJECT data types
 - in columns 14-26
- VariantFormatter class 14-15
- views
 - displaying data in 18-5
 - of data 2-8
- visual components
 - connecting to DataExpress components 5-12

W

- Where clause 11-13
- Where page
 - Data Modeler 11-13