Developing database applications

VERSION 3

Borland®
JBuilder™
for Windows 95, Windows 98, & Windows NT

Borland, A Division of Inprise Corporation
100 Enterprise Way, Scotts Valley, CA 95066-3249

Refer to the file DEPLOY.TXT located in the root directory of your JBuilder 3 product for a complete list of files that you can distribute in accordance with the JBuilder 3 License Statement and Limited Warranty.

Inprise 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.

Printed in the U.S.A.

JBE1330WW21001 2E1R399
9900010203-9 8 7 6 5 4 3 2 1
PDF

Contents

Chapter

1

Developing database applications

This part of the manual provides information on using JBuilder's DataExpress database functionality to develop database applications. It also explains the interrelationships between the main JavaBeans Component Library(JBCL) and dbSwing UI and data components and classes, and how to use them to create your database 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 Borland Online for documentation additions and updates at http://www.borland.com/techpubs/jbuilder.

Visit the database newsgroup on the borland.com Web page at news://forums.inprise.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.

Note All versions of JBuilder provide direct access to SQL data through the JavaSoft JDBC API. Some versions of JBuilder provide additional DataExpress components (on the Data Express tab of the Component Palette) that greatly simplify RAD visual development of database applications, as described in this book.

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

- **Install and set up.**

  Chapter 2, "Installing and setting up JBuilder for database applications" includes the setup required to step through and run the sample applications referenced in this manual. This includes JBuilder setup for access of data through JDBC, JBuilder sample files, and the Local InterBase Server.

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

  Chapter 3, "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.

- **Connect to a database.**

  Chapter 4, "Connecting to a database" describes how to connect your database application to a remote server.

- **Provide data to your application.**

  Chapter 5, "Accessing data" describes how to create a local copy of the data from your data source, and which DataExpress package components to use. All applications which access data need to implement this phase (called *providing*) so that the data is available to your application.

  You might wish to use a data module to hold the DataExpress package components. Chapter 9, "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.

  "Working with columns" on page 5-41 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, how to define the order of display of columns, etc.

  Chapter 8, "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.

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

  Chapter 10, "Persisting and storing data in a DataStore" discusses using *DataStore* components for organizing an application's *StorageDataSet*s, files, and serialized JavaBean/Object state into a single, Pure Java, portable, compact, high-performance, persistent storage. For information beyond the scope of the book, see the *DataStore Programmer's Guide*.

- **Save changes to your data.**

  Chapter 6, "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 8, "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 are often included in database applications, and how you can add them to yours.

  - Chapter 7, "Establishing a master-detail relationship" provides information on linking two or more data sets to create a parent/child (or master-detail) relationship.
  - Chapter 11, "Filtering, sorting, and locating data" provides information on how to filter, sort, and locate data in a data set.
  - Chapter 12, "Adding functionality to database applications" include
    - formatting and parsing data with edit or display patterns
    - creating calculated columns
    - creating a lookup field with a calculated field or with a pick list
    - creating an alternate view of the data
    - creating persistent, or required, fields
  - Chapter 13, "Using other controls and events" includes using JBCL and dbSwing data-aware controls. These include
    - using *dbSwing* components to develop a database user interface
    - data-aware controls and their usefulness to database applications
    - handling errors and exceptions in your application
  - Chapter 16, "Database administration tasks" includes such common database tasks as
    - using the JDBC Explorer to browse and edit data, tables, and database schema
    - creating and deleting tables
    - populating tables with data
    - using the JDBC Monitor to monitor or manipulate JDBC traffic (Enterprise version only)
    - using the Data Migration Wizard to migrate, or move, data between SQL databases and desktop databases

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

- **View a sample database application.**

  Chapter 17, "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 14, "Creating a distributed database application" discusses using DataExpress components in a distributed object computing environment.

  Chapter 15, "Creating database applications with the Data Modeler and Application Generator" provides information on using JBuilder's Data Modeler and Application Generator to create a two-tier (client-server) database application.

Extending this functionality to an n-tier application is discussed in Chapter 1, "Developing distributed applications"in *Developing distributed applications*.

For deploying database applications, you may wish to consider using servlets.

- **Create a servlet.**

  Chapter 10, "Developing servlets" in *Developing distributed applications* describes how to create a servlet in JBuilder, provides a tutorial for practice, and provides links to sample servlets on other Web sites. Servlets are server-side versions of applets, or a server-side Java program that gets initiated when certain HTML is encountered.

Chapter 18, "Database development Q&A", is a collection of answers to selected questions from the borland.com database newsgroup.

Chapter

2

Installing and setting up JBuilder for database applications

To step through and run many of the database tutorials included in this book, you'll need to install these software components:

- JBuilder, JDBC, and the JDBC-ODBC bridge
- JBuilder sample files
- Local InterBase Server

If you have the Enterprise version of JBuilder, you can also install:

- InterClient. InterClient is an all-Java JDBC driver for InterBase. It is available for Solaris, HP-UX, Windows NT, and Windows 95. See "Using InterClient" on page 2-6 for more information.

Installing JBuilder, JDBC, and the JDBC-ODBC bridge

When you install JBuilder, select the Typical option. If you select Custom, you should select the following options:

- Program Files - This option includes installation of JBuilder, JDBC, the JDBC-ODBC bridge, and JBuilder sample files.
- Sample Files - This option installs the data set tutorials and the international demo application.
- Reference Application - This option installs the basic Cliffhanger application. To run this application, you need the JDBC-ODBC bridge and Local InterBase Server.

JavaSoft 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 pure Java implementation that can be executed directly from a local or centralized remote server. JDBC allows for much simpler maintenance and deployment than ODBC.

According to JavaSoft's web site, JDBC is been 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 via the JDBC-ODBC bridge provided by JavaSoft. 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.

Installing JDBC and the JDBC-ODBC bridge

JBuilder DataExpress components are implemented using the JavaSoft database connectivity (JDBC) Application Programmer Interface (API). To create a Java data application, the JavaSoft 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 JavaSoft JDBC-ODBC bridge software. The installation takes care of putting these things in the right place.

The JDBC portion of the setup program installs the classes from the *java.sql* package in the classes.zip file in the \java\lib\ directory.

The JDBC-ODBC bridge portion of the setup program installs the JDBC-ODBC bridge classes in the same classes.zip file. The JdbcOdbc.dll file is installed in the \java\bin directory.

For more information about JDBC or the JDBC-ODBC bridge, visit the JDBC Database Access API Web page at http://www.javasoft.com/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, Delphi, IntraBuilder, Paradox, or Visual dBASE. To do so, look at the underlying database that your application connects to and determine whether the database is a local or remote (SQL) database.

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

- An all-Java JDBC driver, like InterClient, for your server. Some versions of JBuilder include JDBC drivers. Check the Borland Web page (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.
- 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.

The two options when connecting to local, non-SQL databases such as Paradox or Visual dBASE are:

- Use the Data Migration Wizard to move the data to InterBase or another supported database. For information on using the Data Migration Wizard, see "Moving data between databases" on page 16-10.
- Use an ODBC driver appropriate for the table type and level you are accessing in conjunction with the JDBC-ODBC bridge software.

If you encounter any problems connecting to a JDBC database, see the topic "Troubleshooting JDBC database connections in the tutorials" on page 2-8.

Installing JBuilder sample files

The JBuilder samples directory contains files for various tutorials and examples presented in this manual. The Typical setup option installs the sample files to the samples\com\borland\samples\dx directory of your JBuilder installation by default. You can also select the Sample Files option under Custom setup to install the sample files.

The database sample applications use sample data provided with Local InterBase Server.

Installing Local InterBase Server

After installing JBuilder, install Local InterBase. Installing Local InterBase installs both the InterBase Client and InterBase Server on your local machine. The following steps provide the configuration information that is required for all of the tutorials in the database tutorials.

1. Run install.exe from the JBuilder CD, and select Local InterBase. Write down the Certificate ID and Certificate Key numbers. You will need them later.
2. Click Next to move from the InterBase Server Setup dialog, recommending that you exit all Windows programs before running the setup.
3. Read the Installation Information, and click Next to continue.

4 Read the License Agreement, and click the Yes button.

5 Enter your Software Activation Certificate ID and Certificate Key. Click Next to continue.

6 Install InterBase into the default directories specified by the setup program, as recommended in the Local InterBase install.txt. Click Install to install all components.

7 For information on stopping InterBase Server, see "Stopping the InterBase Server" on page 2-5. InterBase Server must be running to complete the database tutorials.

8 Restart your computer. InterBase Server runs at startup.

When the installation is complete, you must create an ODBC Data Source to work with the database tutorials. To create the ODBC Data Source (using Windows NT),

1 From the Control Panel, select ODBC.

2 From the ODBC Data Source Administrator dialog, select the System DSN page.

3 Click the Add button to add a data source.

4 Select "INTERSOLV InterBase ODBC Driver (*.gdb)" from the list of drivers. Click the Finish button.

5 On the ODBC InterBase Driver Setup dialog, enter "DataSet Tutorial" for the Data Source Name. Click OK to close the setup dialog.

6 Click the Configure button.

7 Enter the following values in the InterBase ODBC Configuration dialog to work with the database tutorials:

For this option	Make this choice
Data Source Name	DataSet Tutorial
Description	Optional. You can leave this blank
Network Protocol	<local>
Database	Enter the path to the employee database *employee.gdb*, from your InterBase directory; the default location for this file is `c:\Program Files\InterBase Corp\InterBase\Examples\database\employee.gdb`
Username	SYSDBA
Password	masterkey

8 Click Test Connection to make sure the parameters are correct. The InterBase Server must be running for the connection to be successful. You can start the InterBase server from its program group.

9 Click OK to close the Configuration dialog.

10 Click OK to close the Administrator.

Starting the InterBase Server

By default, InterBase Server and InterBase guardian will run in Windows startup mode. To change to manual startup, select InterBase Configuration Tool from the InterBase program group, and change the Startup Mode to Manual Startup.

Stopping the InterBase Server

Whether InterBase is started automatically when Windows starts up, or if you start it manually from its Program Group, it will be running in the background as a service. To stop InterBase Server from running,

1. Select Services from the Control Panel.
2. Select InterBase Server from the list of services.
3. Click the Stop button.

Tips on using InterBase

- Sample InterBase databases are installed by the setup program. You may want to make a copy of the employee.gdb 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.
      - 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, "Saving changes from a QueryDataSet" on page 6-2.

- One user name and password combination that always works with new InterBase databases is "SYSDBA" (without quotes) as user name and "masterkey" as password. This combination is used in the tutorials in this manual.

Note InterBase passwords are case-sensitive.

- At any time after the setup program has completed, you can create additional ODBC Data Sources by clicking on the ODBC icon from the Control Panel.

To view the metadata for InterBase tables,

1. Make sure you are connected to a database by running InterBase Server.
2. Start InterBase Windows ISQL from the InterBase Program Group.
3. Select File | Connect to Database from the InterBase Interactive SQL menu.
4. Browse to the employee.gdb database in the InterBase examples\database directory for the Database field in the Database Info section of the Database Connect dialog.
5. Enter your user name and password. Click OK to connect.
6. Select Metadata | Show to explore the constraints on the employee and other tables in the sample InterBase databases.

You could also view data and metadata using the JDBC Explorer, available in JBuilder by selecting Tools | JDBC Explorer.

Using InterClient

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.

To install InterClient,

1. Make sure that JBuilder is installed, and has been run at least once, so that the InterClient installation can add the path to InterClient.jar to the Djava.class.path line in the JavaVM_properties section of JBuilder.ini, and add a library entry to library.ini.
2. Close all running applications on your computer, including JBuilder.
3. Start InterBase Server by selecting it from the InterBase Program Group. If you have not installed InterBase, see "Installing Local InterBase Server" on page 2-3.
4. Install InterClient from the JBuilder CD. Run Install.exe from the JBuilder CD, select InterClient on the installation screen, and click OK. Accept all defaults. When installation is complete, you can configure InterClient for your needs. When you are done, restart your computer.

To set up JBuilder for use with InterClient, follow the topic "Setting up InterClient for database tutorials" on page 4-6.

For a tutorial using the InterClient all-Java JDBC driver, see "Tutorial: Connecting to a database using an all-Java JDBC driver" on page 4-6.

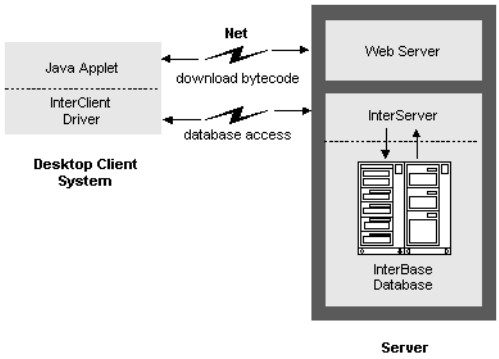
InterClient allows Java applets and applications to:

- Open and maintain a high-performance, direct connection to an InterBase database server
- Bypass resource-intensive, stateless Web server access methods
- Allow higher throughput speeds and reduced Web server traffic

The primary audience for InterClient is developers who wish to create Java-based client-server applications. Java developers should be able to seamlessly swap RDBMS back-ends underneath their JDBC applications. InterClient provides a small footprint, easy to maintain RDBMS (InterBase) as the back end to JDBC applications. An InterBase back end is an ideal solution because it's small, economical, and conforms to the same SQL standards as the JDBC.

InterBase developers who are writing new Java-based client programs can use InterClient to access their existing InterBase databases. Because InterClient is an all-Java driver, it can also be used on Sun's new NC (Network Computer), a desktop machine that runs applets. The NC has no hard drive or CD ROM; users access all of their applications and data via applets down loaded from servers.

The following figure shows the InterClient Architecture:



InterClient consists of two major pieces:

- A client-side Java package, called *InterClient*, containing a library of Java classes that implement most of the JDBC API and a set of extensions to the JDBC API. This package interacts with the JDBC Driver Manager to allow client-side Java applications and applets to interact with InterBase databases.
- A server-side driver, called *InterServer*. This server-side middle ware serves as a translator between the InterClient-based clients and the InterBase database server.

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 via 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.

Troubleshooting JDBC database connections in the tutorials

Connecting to a SQL server using JDBC can result in error messages generated by JDBC. The errors listed below may be encountered when creating the tutorials in this manual and are included to help you troubleshoot connection problems.

Note JDBC errors may be stacked with other warnings and informational text. If errors and messages are stacked, you may need to refer to several lines in the JDBC error response.

Unable to load dll 'JdbcOdbc.dll'

The JdbcOdbc.dll can't be found. Verify that the JdbcOdbc.dll has been installed in a directory that is on your DOS path and that you do not have any older versions of this .DLL file earlier on your path. Normally, this file is installed in \java\bin. If it isn't there, it may have been deleted or the jdbc-odbc bridge was not correctly installed. For installation instructions, see Chapter 2, "Installing and setting up JBuilder for database applications."

java.sql.SQLException: No suitable driver

When connecting to a URL, each registered driver is used to check the specified URL. This error occurs when none indicate support of that URL.

This error can occur when the driver class file for the specified URL cannot be found (for example, sun.jdbc.odbc.JdbcOdbcDriver). Verify that the driver class name is specified correctly and that it has been properly installed. You generally identify the driver name in the connection dialog for a Database component. Select the Choose URL button for a list of known URLs.

Another possible cause of this error is an invalid URL. In this case, verify that the URL name is correctly entered.

Data source name not found

The *DataSource* name that is specified in the application does not match any data sources set up in your ODBC configuration (for example, during the Local InterBase installation). See "Installing Local InterBase Server" on page 2-3 on setting the *DataSource* name.

For a list of data sources, select the Choose URL button in the connection dialog for the Database component. Select the Show data sources button in the ODBC Drivers group.

Connection failed java.sql.SQLException: [...] unavailable database

The database you have specified is not available. The square brackets contain the name of the driver which you are attempting to use. Check that the server is running and available.

If the server is the InterBase Server that is referenced in the tutorials, select InterBase Server from its Program Group to start the InterBase Server.

Chapter

3

Understanding JBuilder database applications

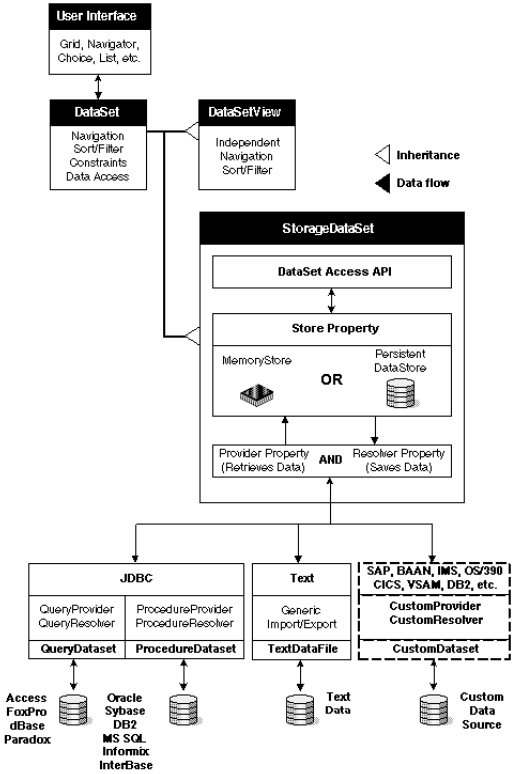
A database application is any application that accesses stored data and allows you to view and perhaps modify or manipulate that data. In most cases, the data is stored in a database. However, data can also be stored in files as text, or in some other format. JBuilder allows you to access this information and manipulate it using properties, methods, and events defined in the *DataSet* packages of the DataExpress Component Library in conjunction with the JavaBeans Component Library (JBCL) and 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 and applets 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 JavaSoft 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 *DataSet*s 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. All of our data-aware JBCL and dbSwing controls have a *dataSet* property. This means a *GridControl* or a *JdbTable* 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 *StorageDataSet*s 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 *DataSet*s, 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 controls need to dynamically switch to a new *DataSet*. The controls 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. *TableDataSet*s, *Column*s, and data can be added through *DataSet* methods or by importing data with a *DataFile* component like *TextDataFile*.

The next section 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 to two interfaces, 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 popular databases such as support for Oracle, Sybase, Informix, InterBase, DB2, MS SQL Server, Paradox, dBASE, FoxPro, Access, and other popular databases. In the future, borland.com and third parties can 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 a pure 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 such as grid, list, field, navigation via point and click property settings made in a visual designer. JBuilder ships with Java JFC based visual components that bind directly to *DataSet* components.

The benefits of using the modular DataExpress architecture include:

- *Network computing*. As mentioned, the provider/resolver approach isolates interactions with arbitrary data sources to two clean points. There are two other benefits to this approach:
  - The provider/resolver can be easily partitioned to a middle tier. Since provider/resolver logic typically has a transactional nature, it is ideal for partitioning to a middle tier.
  - It is a "stateless" computing model that is ideally suited to network computing. The connection between the *DataSet* component client and the data source can be disconnected after providing. When changes need to be saved back to the data source, the connection need only be re-established for the duration of the resolving transaction.
- *Rapid development of user interfaces*. Since *DataSet*s can be bound to data-aware components with a simple property setting, they are ideally suited for rapidly building database application user interfaces.
- *Mobile computing*. With the introduction of the *DataStore* component, DataExpress applications have a persistent, portable database. The *DataStore* can contain multiple *DataSet*s, arbitrary files, and Java Objects. This allows a complete application state to be persisted in a single file storage. *DataSet*s 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.com Web site at http://www.borland.com/jbuilder/ for a white paper on this topic.

borland.com 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. These packages have been renamed since JBuilder 2.0. Use the Package Migration Wizard (available from Wizards | Package Migration) to update applications written prior to JBuilder 3.0.

The main classes and components in the *com.borland.dx.dataset*, *com.borland.dx.sql.dataset*, and *com.borland.datastore* 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 *Database* 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 *ConnectionDescriptor* object stores connection properties of user name, password, and connection URL. Accessed using the *connection* property.
DataSet	An abstract class that provides basic data set behavior, *DataSet* 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 *DataSet*.	The *SortDescriptor* object contains properties that affect the order in which data is accessed and displayed in a UI control. Set using the *sort* property. See "Sorting data" on page 11-7 for a tutorial. The *MasterLinkDescriptor* object contains properties for managing master-detail relationships between two *DataSet* components. Accessed using the *masterLink* property on the detail *DataSet*. See Chapter 7, "Establishing a master-detail relationship," for a tutorial.

Component/Class	Description	Frequently used properties
StorageDataSet	An abstract class that extends the *DataSet* class by providing implementation for storage of the data and manipulation of the structure of the *DataSet*. You fill a *StorageDataSet* component with data by extracting information from a remote database (such as InterBase or Oracle), or by importing data stored in a text file. This is done by instantiating one of its subclasses: *QueryDataSet*, *ProcedureDataSet*, or *TableDataSet*.	The *tableName* property specifies the data source of the *StorageDataSet* component. The *maxRows* property defines the maximum number of rows that the *DataSet* can initially contain. The *readOnly* property controls write-access to the data.
DataStore	The *DataStore* component provides high performance data caching and compact persistence for DataExpress *DataSet*s, 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. See Chapter 10, "Persisting and storing data in a DataStore," and the *DataStore Programmer's Guide*, for more description of the *DataStore* component.	Caching and persisting *StorageDataSet*s in a *DataStore* is accomplished through two required property settings on a *StorageDataSet* called *StorageDataSet.store* and *StorageDataSet.storeName*. By default, all *StorageDataSet*s use a *MemoryStore* if the *StorageDataSet.store* property is not set. Currently *MemoryStore* and *DataStore* are the only implementations for the *store* property. The *StorageDataSet.storeName* property is the unique name associated with this *StorageDataSet* in the *DataStore* directory.
DataStoreDriver	*DataStoreDriver* is the JDBC driver for the *DataStore*. 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.	
QueryDataSet	The *QueryDataSet* component stores the results of a query string executed against a server database. This component works with the *Database* component to connect to SQL server databases, and runs the specified query with parameters (if any). Once the resulting data is stored in the *QueryDataSet* component, you can manipulate the data using the *DataSet* API. See "Querying a database" on page 5-13 for more description and a tutorial using this component.	The *QueryDescriptor* object contains the query statement, query parameters, and database component. Accessed using the *query* property.
ProcedureDataSet	The *ProcedureDataSet* component holds the results of a stored procedure executed against a server database. This component works with the *Database* component in a manner similar to the *QueryDataSet* component. See "Obtaining data through a stored procedure" on page 5-30 for more description and a tutorial using this component.	The *ProcedureDescriptor* object contains the SQL statement, parameters, database component, and other properties. Accessed using the *procedure* property of the *ProcedureDataSet* component.

Component/Class	Description	Frequently used properties
TableDataSet	Use this component when importing data from a text file. This component extends the *DataSet* class by adding behavior that allows it to mimic SQL server functionality, without requiring a SQL server connection. See Chapter 8, "Importing and exporting data from a text file" for more description and a tutorial using this component.	The (inherited) *dataFile* property specifies the file name from which to load data into the *DataSet* and to save the data to.
DataSetView	This component presents an alternate "view" of the data in an existing *StorageDataSet*. It has its own (inherited) *sort* 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 *StorageDataSet*. See "Presenting an alternate view of the data" on page 12-2 for more description and a tutorial using this component.	The *storageDataSet* property indicates the component which contains the data that the *DataSetView* presents a view of.
Column	A *Column* represents the collection from all rows of a particular item of data, for example, all the *Name* values in a table. A *Column* gets its value when a *DataSet* is instantiated or as the result of a calculation. The *Column* is managed by its *StorageDataSet* component. See "Working with columns" on page 5-41 for more description and a tutorial using this component.	You can conveniently set properties at the *Column* level so that settings which affect the entire column of data can be set at one place, for example, *font*. JBuilder design tools include access to column-level properties by double-clicking any *StorageDataSet* in the Component tree, then selecting the *Column* that you want to work with. The selected *Column* component's properties and events display in the Inspector and can be edited there.
DataRow	The *DataRow* component is a collection of all *Column* data for a single row where each row is a complete record of information. The *DataRow* component uses the same columns of the *DataSet* it was constructed with. A *DataRow* is convenient to work with when comparing the data in two rows or when locating data in a *DataSet*. It can be used in all *DataSet* methods that require a *ReadRow* or *ReadWriteRow*.	

Component/Class	Description	Frequently used properties
ParameterRow	The *ParameterRow* component has a *Column* for each column of the associated data set that you may want to query. Place values you want the query to use in the *ParameterRow* and associate them with the query by their *ParameterRow* column names. Although scoped *DataRows* can be used for parameter settings, the *Column* components in the *DataRow* map directly to the column and therefore do not allow more than one reference in a *DataRow*. See "Using parameterized queries to obtain data from your database" on page 5-23 for more description and a tutorial using this component.	
DataModule	The *DataModule* is an interface in the *com.borland.dx.dataset* package. A class that implements *DataModule* 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 component tree, you place into it various *DataSet* objects, and provide connections, queries, sorts, and custom business rules logic. Data modules simplify reuse and multiple use of collections of *DataSet* components. For example, one or more UI classes in your application can use a shared instance of your custom *DataModule*. See Chapter 9, "Using data modules to simplify data access" for more description and a tutorial using this component.	

There are several 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 Library Reference* documentation.

Chapter

4

Connecting to a database

The *Database* component handles the JDBC connection to a SQL server and is required for all database applications involving server data. JDBC is the JavaSoft Database Application Programmer Interface, a library of components and classes developed by JavaSoft 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. JDBC drivers can either be entirely written in Java so that they can be downloaded as part of an applet, or they can be implemented using native methods to bridge to existing database access libraries. For more information about JDBC, visit the JavaSoft JDBC Database Access API Web page at http://www.javasoft.com/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. (See "Installing JBuilder, JDBC, and the JDBC-ODBC bridge" on page 2-1.)

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 of the driver options that may ship with JBuilder are:

- **DataStoreDriver**

  DataStoreDriver is the JDBC driver for the *DataStore*. 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.

- **InterClient**

  As an all-Java JDBC driver for 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. See "Using InterClient" on page 2-6 or "Tutorial: Connecting to a database using an all-Java JDBC driver" on page 4-6 for more information.

You can connect JBuilder applications to remote or local SQL databases, or to databases created with other Borland applications such as C++ Builder, Delphi, IntraBuilder, Paradox, or Visual dBASE. To do so, look at the underlying database that your application connects to and determine whether the database is a local or remote (SQL) database.

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

- An all-Java JDBC driver, like InterClient, for your server. Some versions of JBuilder include JDBC drivers. Check the Borland Web page (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. The "Tutorial: Connecting to a database using an all-Java JDBC driver" on page 4-6 describes how to connect to a database using InterClient as your all-Java JDBC driver.
- An ODBC-based driver for your server that you use with the JDBC-ODBC bridge software. The "Tutorial: Connecting to a database using the JDBC-ODBC bridge" on page 4-3 describes how to connect to a database using the JDBC-ODBC bridge, and InterBase.

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.

The two options when connecting to local, non-SQL databases such as Paradox or Visual dBASE are:

- Use the Data Migration Wizard to move the data to InterBase or another supported database. For information on using the Data Migration Wizard, see "Moving data between databases" on page 16-10.

- Use an ODBC driver appropriate for the table type and level you are accessing in conjunction with the JDBC-ODBC bridge software.

If you encounter any problems connecting to a JDBC database, see the topic "Troubleshooting JDBC database connections in the tutorials" on page 2-8.

Tutorial: Connecting to a database using the JDBC-ODBC bridge

This tutorial assumes you are familiar with the JBuilder design tools. For more information on these tools, see the topic "JBuilder's visual design tools" in the *Building Applications with JBuilder* section of this book.

This tutorial outlines:

- Adding a Database component to your application
- Setting Database connection properties
- Using the Database component in your application

For information on connecting to a database using an all-Java JDBC driver, see "Tutorial: Connecting to a database using an all-Java JDBC driver" on page 4-6.

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 these components to an instantiated *Database* component. Multiple data sets can share the same database, and often will.

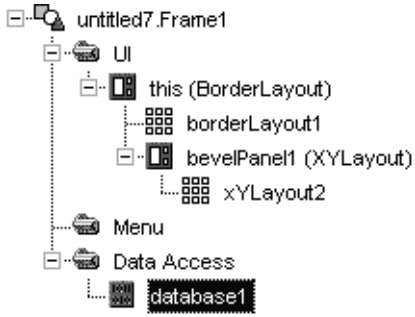
To add the *Database* component to your application,

1. Create a new project and application files using the Project and Application Wizards. (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 all defaults.
2. Open the UI Designer by highlighting the Frame1.java file in the Navigation pane, then selecting the Design tab at the bottom of the AppBrowser.
3. Select the Data Express tab from the Component Palette. Click the *Database* component.

4 Click anywhere in the Component Tree 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 Component tree:



Setting Database connection properties

The *Database connection* property specifies the JDBC connection URL, User Name, Password, and optional JDBC driver(s). 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. A connection to a server must be established to successful set these properties.

You can access the *ConnectionDescriptor* object programmatically or you can set connection properties through the user interface. If you access the *ConnectionDescriptor* programmatically, follows these guidelines:

- If you set *promptPassword* to **true**, you should also call *openConnection()* for your database.
- When you call *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 control or data set first needs data, but the application may hang at this point.

The following steps describe how to set connection properties through the user interface, and provide the resulting code.

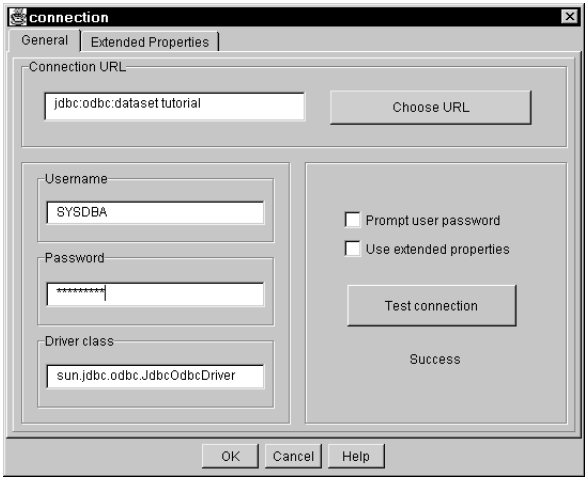
1 Make sure that the server is started, for example, start InterBase Server by selecting it from the InterBase program group.

2 Select the *Database* object in the Component Tree. Double-click the *connection* property in the Inspector window to open the *connection* property editor.

3 Set the following properties:

Property	Description
Connection URL	The Universal Resource Locator (URL) of the database, for example, `jdbc:odbc:DataSet Tutorial`. Select the Choose URL button to select an URL from a list of previously selected URLs or available data sources.
Username	The user name authorized to access the server database, for example, `SYSDBA`.
Password	The password for the authorized user, for example, `masterkey`.
Prompt user for password	Whether to prompt the user for a password each time.
Driver Class	The class name of the JDBC driver that corresponds to the URL, for example, `sun.jdbc.odbc.JdbcOdbcDriver`.

4 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. When successful, the dialog looks like this:



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 like this:

```
database1.setConnection(new
 com.borland.dx.sql.dataset.ConnectionDescriptor("jdbc:odbc:DataSet
    Tutorial", "SYSDBA", "masterkey", false, "sun.jdbc.odbc.JdbcOdbcDriver"));
```

Tip Once a database URL connection is successful, you can use the JDBC Explorer 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. For more information on using the JDBC Explorer, see "Exploring database tables and metadata using the JDBC Explorer" on page 16-1.

Using the Database component in your application

Now that your application includes the *Database* component, you'll want to use a component that needs it. 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 5-13
- "Using parameterized queries to obtain data from your database" on page 5-23
- "Obtaining data through a stored procedure" on page 5-30

If you work with a complex data model or need to change data components in your application, consider encapsulating 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 9, "Using data modules to simplify data access".

Tutorial: Connecting to a database using an all-Java JDBC driver

Before connecting to an SQL database, you must have an appropriate driver installed. For this example we assume that InterClient, an all-Java JDBC driver that ships with JBuilder, has been installed. If it has not, see "Using InterClient" on page 2-6. The first section of this tutorial discusses setting up InterClient for use with JBuilder. The second section discusses connecting to a SQL database from JBuilder using InterClient.

The JDBC Explorer has commonly used connection URL and driver names. Select Tools | JDBC Explorer and access its online help for more information.

Setting up InterClient for database tutorials

To set up InterClient for use with JBuilder,

1. Run InterBase Server by selecting it from the InterBase Program Group.
2. Run InterServer by selecting it from the InterBase InterClient Program Group. When InterServer is running, its icon will display in the Task Bar.
3. Start JBuilder. Close all projects.
4. Add InterClient to the list of available Java libraries. To do this, select Project | Default Properties, and set the following options:
   1. Click the Add button beside Java Libraries.
   2. Click New in the Select A Java Library To Add dialog box.

3. In the Select A Java Library To Add dialog, set the following properties:
   - Name: InterClient
   - Class Path: Use the Browse button to locate the file InterClient.jar. By default, this file is located at C:\Program Files\InterBase Corp\InterClient\InterClient.jar
4. Click OK.
5. Click OK.

5. If you installed InterClient before JBuilder was run for the first time, edit JBuilder.INI and add the path to InterClient.jar to the Djava.class.path line in the JavaVM_properties section, following existing syntax.

Now JBuilder and InterClient are set up to work together. When developing and running applications using the JDBC driver in the future, you will only need to start InterServer to have access to the JDBC driver.

The next section describes adding a *Database* component to your application and setting its *connection* properties using the JDBC driver.

Using InterClient all-Java JDBC drivers in JBuilder

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 a local or remote database. Once a database connection has been established, you can use *QueryDataSet* and/or *ProcedureDataSet* components to run queries or stored procedures against one or more database tables, and you can use visual controls, such as a *GridControl*, to display the results.

To add the *Database* component to your application and connect to a database,

1. Start InterServer by selecting it from the InterClient program group or from the Start menu. When running, its icon will display in the Task Bar.
2. Select File | Close, or skip this step to add data connectivity to an existing project and application.
3. Create a new project and application files by selecting File | New, and double-clicking the Application icon. Select all defaults.
4. Open the UI Designer by highlighting the Frame1.java file in the Navigation pane, then selecting the Design tab at the bottom of the AppBrowser. In this application, user interface elements will go in the Frame file.
5. Select the Data Express tab from the Component Palette. Click the *Database* component, and click anywhere in the Component tree window to add the *Database* component to your application.

6 Set the *Database connection* property to specify the JDBC connection URL, User Name, Password, and JDBC driver(s).

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 Select the *Database* object in the Component tree. Double-click the *connection* property in the Inspector window 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
Connection URL	`jdbc:interbase://localhost/c:/program files/interbase corp/interbase/examples/database/employee.gdb`. In this example, enter the path to the InterBase installation and sample files as they were installed on your computer. The path listed would be appropriate for most default installations. In the future, you can click the Choose URL button to select an URL from a list of previously selected URLs or available data sources.
Username	SYSDBA
Password	masterkey
Driver Class	interbase.interclient.Driver

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.

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 JDBC Explorer 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. For more information about the JDBC Explorer, see "Exploring database tables and metadata using the JDBC Explorer" on page 16-1.

The next step in building your database application would be to use queries and/or 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 5-13
- "Using parameterized queries to obtain data from your database" on page 5-23
- "Obtaining data through a stored procedure" on page 5-30

If you work with a complex data model or need to change data components in your application, consider encapsulating 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 9, "Using data modules to simplify data access".

Chapter

5

Accessing data

The first step in creating a data application is to access information stored in the data source and create a copy so that the data can be manipulated locally by your application. 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.

You may wish to browse and/or edit database data and/or metadata before proceeding with developing your application. You can use the JDBC Explorer to:

- Browse database schema objects.
- View, create, and modify database URLs.
- Create, view, and edit data in existing tables.
- Enter and execute SQL statements.

See the "JDBC Explorer" online help topic for more information on using the JDBC Explorer.

By default, all data sets store row data in memory (*MemoryStore*). Consider using a *DataStore* component, instead. *DataStore* is a high-performance, small-footprint, 100% Pure Java™ multi-faceted 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*. See the *DataStore Programmer's Guide* for more information on using *DataStore*s.

There are several ways to access data in JBuilder. The rest of this topic focuses on using JBuilder's DataExpress architecture to provide data to an application. You could also create a database application using the Data Modeler and Application

Generator. See Chapter 15, "Creating database applications with the Data Modeler and Application Generator" for information on how to do this.

To access data in JBuilder using the DataExpress components, add one of the following *StorageDataSet* subclasses to your application or to a data module:

- *TextDataFile*

  "An introductory database tutorial using a text file" on page 5-3 steps you through creating a database application, even if you are not connected to any SQL or desktop databases. This tutorial uses data in a text file that ships with JBuilder.

  This tutorial also steps through building a user interface for the application using JBuilder design tools.

- *QueryDataSet*

  "Querying a database" on page 5-13 steps through creating the local copy of the data by executing a query and storing its results in a *QueryDataSet* component.

  "Using parameterized queries to obtain data from your database" on page 5-23 outlines the steps required when adding parameters to your query statement.

- *ProcedureDataSet*

  "Obtaining data through a stored procedure" on page 5-30 steps through creating the local copy of the data by executing a stored procedure and storing its result set in a *ProcedureDataSet* component.

- *TableDataSet*

  Chapter 8, "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*

  "Writing a custom data provider" on page 5-38 discusses custom data providers, and how they can be used as providers for a *TableDataSet* and any *DataSet* derived from *TableDataSet*.

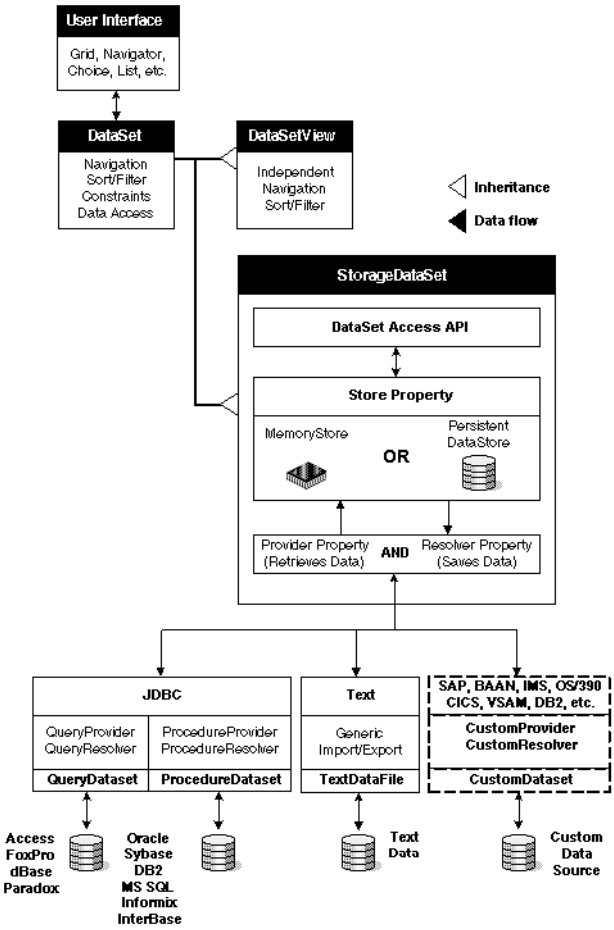
For *QueryDataSet* and *ProcedureDataSet* components, the data source is often a SQL server database. In this case, you also need a *Database* component to handle the connection to the server. See Chapter 4, "Connecting to a database" for more information on connecting to a server. When using the *TableDataSet* 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.

The *Column* component stores important column-level metadata type properties (such as data type and precision) as well as visual properties such as font and alignment. For *QueryDataSet* and *ProcedureDataSet* components, a set of *Column* components is dynamically created each time the *StorageDataSet* is instantiated, mirroring the actual columns in the data source at that time. See the topic "Working with columns" on page 5-41 to learn more about creating persistent columns and setting column properties. See Chapter 8, "Importing and exporting data from a text

file" to learn how to programmatically add *Column* components and read in data, such as date and timestamp data, for which there are no standards.

The following diagram illustrates the different ways to obtain and store data in a JBuilder component:

Figure 5.1 DataExpress Architecture



For more information on JBuilder's DataExpress architecture, see "Understanding JBuilder's DataExpress architecture" on page 3-3.

An introductory database tutorial using a text file

This introductory tutorial creates an application that reads data from a text file, and steps through building the user interface (UI) for the application using JBuilder design tools. Then it explores the JBuilder DataExpress architecture and applies

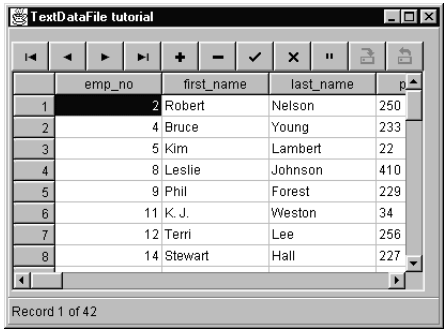
database concepts to the tutorial even though this application does not connecting to a true database.

The UI consists of three controls: a grid control that displays the data from the text file, a navigator to aid in browsing through the data, and a status area that displays messages.

The basic steps to completing this sample application are

1. "Creating the application structure" on page 5-4 describes using the Project and Application wizards to create the basic application files.
2. "Adding UI components to your application" on page 5-6 explains how to add the *GridControl* and *NavigatorControl* to your application. To create a UI using dbSwing components, see "Creating a database application UI using dbSwing components" on page 13-2.
3. "Adding DataExpress components to your application" on page 5-9 explains how to add the *TableDataSet* and *TextDataFile* components which do not have a visual representation at run time.
4. "Setting properties to connect the components" on page 5-9 describes how to get the components "talking" to each other by setting the appropriate properties.
5. "Compiling, running, and debugging a program" on page 5-12 explores getting your application to run and do what you want it to do.

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



You can see the code for the completed tutorial by opening the sample project file samples\com\borland\samples\dx\TextDataFile\TextDataFile.jpr of your JBuilder installation.

Creating the application structure

Let's start by creating the application using the Project and Application Wizards. To start the Project Wizard,

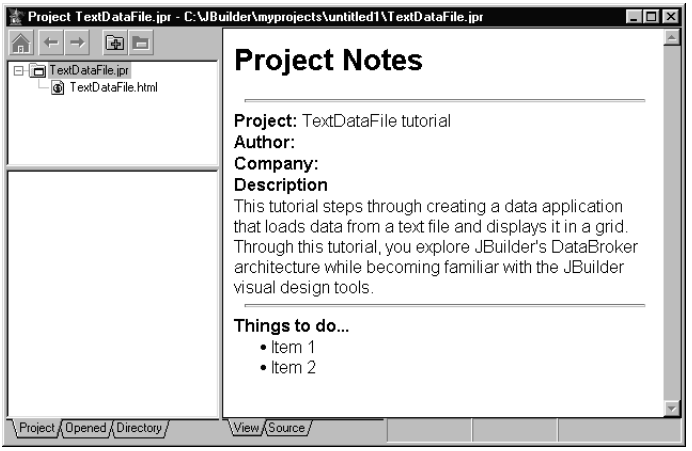
1. Choose File | Close from the JBuilder menu to close any existing projects.
2. Choose File | New Project to start the Project Wizard.

3 Keep the specified path, and change the default project name to `TextDataFile.jpr` by clicking the Browse button to the right of the File field and entering `TextDataFile.jpr` in the File Name field. Click Save to exit this dialog.

4 Optionally, enter information for a title (TextDataFile tutorial), your name, your company, and a project description (This tutorial...).

5 Click Finish.

The AppBrowser window displays the project's structure:



Now that the basic project structure is created, let's start the Application Wizard to create a new Java application shell which contains a *Frame*.

1 Select the File | New menu option, then double-click the Application icon. The first page of the Application Wizard appears.

2 Leave the default package name as is.

3 Enter `TextDataFileApp` in the Class field of the Application Class box.

4 Uncheck Use Only JDK & Swing Classes.

5 Click the Next button.

6 Enter `TextDataFileFrame` in the Class field of the Frame Class box.

7 Enter `TextDataFile Tutorial` in the Title field of the Frame Style box.

8 Place a check beside the Generate Status Bar and Center Frame On Screen options. Leave the remaining options (menu bar, tool bar, and generate about box) at their default values; they are not needed for this example.

9 Click Finish.

In the Navigation pane (top left pane) of the AppBrowser, the TextDataFileApp.java and TextDataFileFrame.java files are added to the project.

Adding UI components to your application

The JBuilder UI Designer is used to build the user interface portion of this application. This step adds two UI elements to the application: a *GridControl* and a *NavigatorControl*. The *GridControl* is used to display two-dimensional data, in a format similar to a spreadsheet. The *NavigatorControl* is a set of buttons that help you navigate through the data displayed in a control such as a *GridControl*.

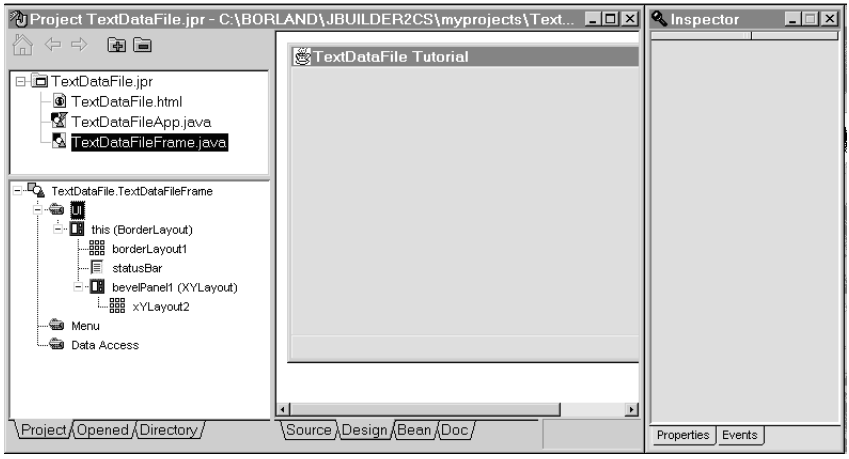
Although this topic shows how to create a UI for your application using JBCL components, you could instead use dbSwing components to create the UI. See "Creating a database application UI using dbSwing components" on page 13-2 for information on creating the UI with dbSwing components.

Adding a UI component

To add a UI component to your application, first display the UI design tools, as follows:

1. Select the `TextDataFileFrame.java` entry in the Navigation pane of the AppBrowser.
2. Click the Design tab at the bottom of the Content pane (the large pane to the right).
3. Select bevelPanel1 in the Component tree. In the Inspector, change *layout* to *XYLayout*.

The UI Designer displays in the Content pane (the large pane on the right side of the AppBrowser), the Component tree displays in the Structure pane (smaller pane on the bottom left), and the Inspector displays in a separate window to the right.



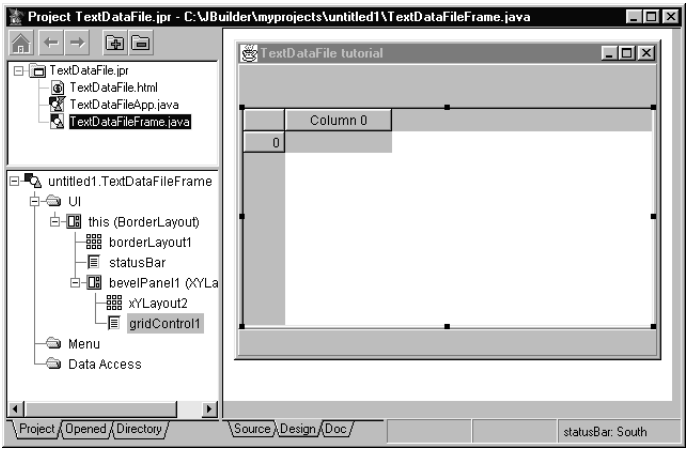
The Navigation pane allows you to easily and quickly select a file. Based on your selection, the other panes of the AppBrowser update as appropriate. The Component tree, for example, updates to display the elements in the selected file. Use the Component tree to navigate through these elements. The Source/Design panes also

update to display the contents of the selected item. To add a UI component to your application,

1. Click the JBCL tab of the Component palette to see the various controls.
2. Click the *GridControl* component:

3. Click where you want the upper left corner of the *GridControl* to start in the UI Designer window and hold the mouse button while you draw the *GridControl* to the desired size, ending at the lower right corner. The rectangular area displayed in the UI Designer is the default size of your application UI. You'll notice a larger rectangular area with a long thin rectangle beneath it. To keep this tutorial simple, draw your *GridControl* within the bounds of the larger shaded area. The long thin rectangle is the status bar created by the Application Wizard.

Your UI Designer window should look similar to this:



Tip To resize a UI component, click the mouse on the component to select it. When a component is selected, a border for the component becomes visible with small squares on the edges of the borders—these are the component's "handles". Move the mouse over any handle; when the cursor changes to a double arrow, click the mouse and drag to resize the component.

Tip To change the alignment of a UI component, right-click on the component to bring up the context menu. You can choose to right, center, or left align the component within the UI Design window. You can also choose to top, middle, or bottom align the component within the UI Design window.

Notice that the Component tree in the lower left pane of the AppBrowser has a new entry: *gridControl1*. This is the *GridControl* you just placed in your application.

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

1. The TextDataFileFrame.java entry should already be highlighted in the Navigation pane (upper left pane of the AppBrowser); if not, select it by clicking it. You'll see the following code that creates the instance of the *gridControl1* object:

   ```
   GridControl gridControl1 = new GridControl();
   ```

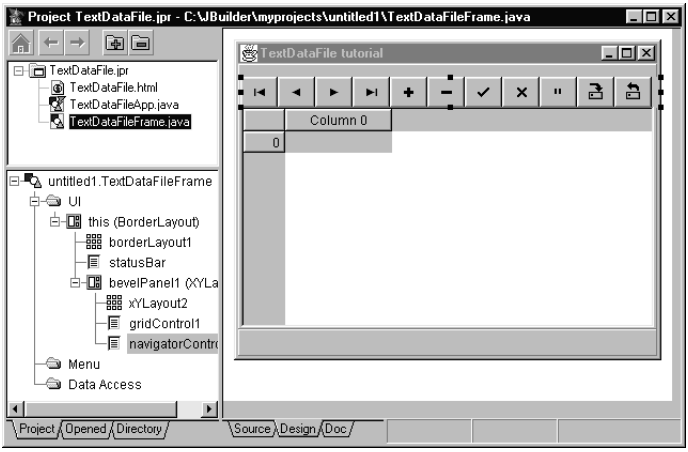
2. Click the *jbInit()* method in the Structure pane; you'll see the following code, which adds the *GridControl* to the application:

   ```
   bevelPanel1.add(gridControl1, new XYConstraints(4, 59, 385, 252));
   ```

   The numbers you see in the *XYConstraints()* method will likely differ as these are the size coordinates for the *GridControl* displayed above. Your coordinates will correspond to the size of the component you draw.

Switch back to the Design window (click the Design tab) and add a *NavigatorControl* just above the *GridControl*, following the steps used to add the *GridControl*. The *NavigatorControl* is located on the JBCL tab of the Component palette, and looks like this:

Your application should look similar to this:



You've just finished creating the UI for your application. Before continuing, select the File | Save All menu option to save your work. The next step is to add DataExpress components and hook them up to the controls that were just created.

For more information on the fine points of UI design, see "Designing a user interface" in *Building Applications with JBuilder*.

Adding DataExpress components to your application

With DataExpress components, you have the choice of dropping them on the UI Designer as with visual controls, or you can drop them directly on the Component tree. The only indication that the DataExpress component is added is the reference to an instance of that object which appears in the Tree (and of course the source code generated in the Content pane); DataExpress components have no visual representation in the UI.

This application requires two DataExpress components:

- a *TableDataSet* component that can read a table (rows and columns) of data from some data source and work with it in memory in the JBuilder *DataSet* format.
- a *TextDataFile* that connects the *TableDataSet* to a particular text file on disk.

These components are found on the Data Express tab of the Component palette. To add the *TableDataSet* DataExpress component to your application,

1 Click the Data Express tab.

2 Click this button to select the *TableDataSet* component:

3 Click in the application design window or in the Component tree to add this component to your application. An entry for an instance of a *TableDataSet* object appears in the DataExpress folder of the Component tree as `tableDataSet1`.

Now add the *TextDataFile* component to your application following the steps above. The button for the *TextDataFile* is:

and once added, appears in the Component tree as *textDataFile1*.

All the components have been added to the application. Before continuing, select the File | Save All menu option to save your work. The next step is to connect them together by setting their properties.

Setting properties to connect the components

Now all the pieces needed for the application are in place, but they're distinct pieces that don't yet "talk" to each other. Setting the appropriate component properties will connect the pieces. Properties are set from the Properties tab of the Inspector. The Inspector window displays the properties for the component selected in the UI Designer window (for UI components) or the Component tree (for both UI and DataExpress components).

Setting properties of DataExpress components

The first step is to get the two DataExpress components "talking" to each other. This is accomplished by setting the *fileName* property of the *TextDataFile* component. This tells JBuilder where to find the text file that contains the data to be read into *textDataFile1*. Since *TextDataFile* is a DataExpress component, select it from the Component tree rather than the UI Designer window.

To set the *fileName* property of the *TextDataFile* component,

1. Click the *textDataFile1* object in the Component tree. The Inspector displays the properties for this component.

Note Make sure the Properties page is selected in the Inspector, not the Events page.

2. Double-click the edit area next to the *fileName* property in the Inspector. It changes color to show that it is active for editing.
3. Click the ellipsis button to display the File Name dialog.
4. Click the Browse button to display the Open dialog. Click the parent directory icon to select the path of samples\com\borland\samples\dx\TextDataFile. Select the employee.txt file. Click the Open button to close the dialog.
5. Click the OK button to close the File Name dialog. The resulting source code in the Source pane that sets this property is similar to:

```
textDataFile1.setFileName("c:\\JBuilder\\samples\\com\\borland\\samples\\
 dx\\TextDataFile\employee.txt");
```

Use *Alt+Z* to toggle the Source window larger and smaller. If you installed JBuilder in a directory other than the one listed above, change the code to reflect the location of the sample files on your computer.

Note Do not use a different text file than employee.txt at this time. Later examples (see Chapter 8, "Importing and exporting data from a text file") show you how to work with your own data files.

Next, connect *tableDataSet1* to the *textDataFile1* component.

1. Select the Design tab.
2. Select *tableDataSet1* in the Component tree. The Inspector displays the properties for this component.
3. Click the area beside the *dataFile* property. Click the down arrow and select the entry for *textDataFile1*. This generates the following source code:

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

4. Save your work using the File | Save All menu option.

The employee.txt file works with the default settings stored in the *TextDataFile* component for properties such as *delimiter*, *separator*, and *locale*. If it required anything other than default values, the appropriate properties could be set at this point.

Now that the DataExpress components are connected to each other, the UI components can be connected.

Setting properties of UI components

The next step is to work with the two UI components, *GridControl* and *NavigatorControl*, as well as the *StatusBar* control that the wizard added for us. To work with these components, select them in either the UI Designer or the Component tree. You can tell when a UI component is selected in the UI Designer by the presence of borders and handles around the component.

To set the *dataSet* property of the *GridControl* component and connect the UI control to live data,

1. On the Design page, select the *GridControl* component.
2. Click the edit area beside the *dataSet* property in the Inspector.
3. Click the down arrow that appears.
4. Select *tableDataSet1* from the drop-down list. This list contains all instantiated *DataSet* components (of which there is only one in this example).

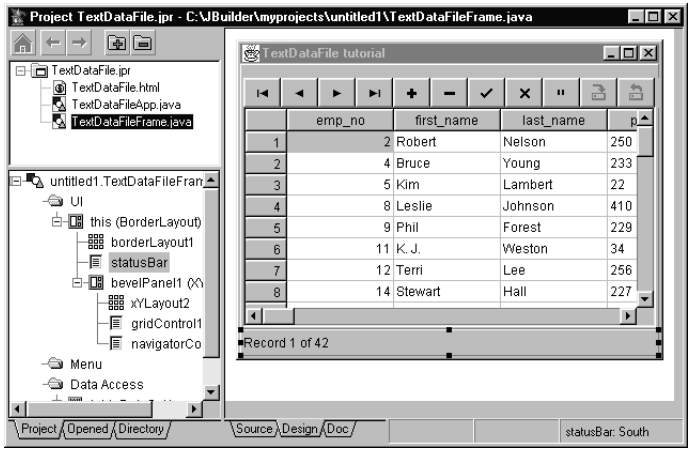
Setting this property fills the *GridControl* with data from the text file. You should see live data in the UI Designer.

Also, set the *dataSet* properties of both the navigator and the status bar to *tableDataSet1*, following the steps as above. Then save your work.

When you set the *dataSet* property of the navigator, the two right-most buttons become dimmed and unavailable, because those buttons do not apply to *DataSet* objects whose data source is not a SQL server database.

When you set the *dataSet* property of the status bar, it displays the current row position and the row count. You cannot navigate the data in the UI Designer; but when you run the application, the status will update as you move the cursor through the data in the grid.

Your application should look like this:



Compiling, running, and debugging a program

Although you are in the UI Designer and have not run the program yet, data appears in the *GridControl* and your application looks complete. But your users won't be using your application in the JBuilder UI Designer. So, the next step is to compile, run, test, and possibly debug the application.

If you need to do so, review the chapters "Compiling Java programs" and "Debugging Java programs" in the online document *Building Applications with JBuilder*. These chapters discuss the two topics in much greater depth. To compile and run the application, click the Run button.

Note If you click the Run button when your program has not been compiled, JBuilder automatically compiles your source and, if no compile errors are generated, runs your application.

If there are syntax errors, the program is not automatically run. If syntax errors are found, an Error pane containing error messages appears in the lower right of the AppBrowser window. Compiler or syntax errors tend to be easier to correct than logic errors. Logic errors are most noticeable at runtime when the application doesn't quite do what you'd like it to.

In this simple application, all properties were selected from drop-down lists or browsers, so syntax errors shouldn't be encountered. Had there been a need for manual typing, for example, a directory path and file name or custom code, chances would be greater for a syntax error to occur.

If all goes well, a message indicating a successful compile appears in the message line at the bottom of the Main window, and the application's UI displays.

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

- The *GridControl* is filled with data from the text file and looks identical to how it is displayed in the UI Designer.
- The *StatusBar* along the bottom of your application displays the number of rows in the grid, and your current row position. The *StatusBar* displays messages generated by the *TableDataSet*. For example, when you change data in the grid, the status bar displays "Editing row". As you move through the rows, you'll notice the *StatusBar* updates automatically.
- You can use the keyboard to navigate the data in the grid.
- Alternatively, you can click the *NavigatorControl* buttons to move you through the data in the grid. This occurs because the grid and the navigator are set 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.
- You can make any changes you want to the data, including changing existing information, or deleting and adding new rows.

- No special functionality has been added to update information back to the data source, so you cannot save changes you make to the data back to the employee.txt file.

Note Although the *NavigatorControl* includes a Save button, this button is dimmed for file-based data sources such as employee.txt. If this application had connected to a true database, the Save button would be available to save changes back to its database source. For more information on saving changes to a database, see "Saving changes from a QueryDataSet" on page 6-2. For more information on saving changes to a data file, see "Exporting data" on page 8-5.

Summary

This program reads data from a text file, displays the data in a *GridControl* for viewing and editing, displays status messages to a *StatusBar*, and includes a *NavigatorControl* component to help browse though the data quickly.

After completing this application, you may want to explore other examples where more complex tasks are explored, showing how JBuilder presents warnings and errors, and working with additional UI and DataExpress components. See Chapter 11, "Filtering, sorting, and locating data" for information on filtering data, locating data, and sorting data.

Querying a database

A *QueryDataSet* component is a JDBC-specific *DataSet* that manages a JDBC provider of data, as provided in the *query* property. Using a *QueryDataSet* component in JBuilder, you can extract data from a server into a data set. This action is called "providing." Once the data is provided, you can view and work with the data locally in data-aware controls. You can store your data to local memory (*MemoryStore*) or to a local single-file database with a hierarchical directory structure (*DataStore*). When you want to save the changes back to your database, you must resolve the data. This process is discussed in more detail in Chapter 3, "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 provided programmatically, or by using JBuilder design tools.

- *Database* component

  The *Database* component encapsulates a database connection through JDBC to the SQL server and also provides lightweight transaction support.

- *QueryDataSet* component

  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* object

  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. 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
database	Specifies what *Database* connection object to run the query against.
query	A Java String representation of a SQL statement (typically a select statement).
parameters	An optional *ReadWriteRow* from which to fill in parameters, used for parameterized queries.
executeOnOpen	Causes the *QueryDataSet* to execute the query when it is first opened. This is useful for presenting live data at design time. You may also want this enabled at run time.
loadOption	An optional integer value that defines the method of loading data into the data set. Options are: **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.

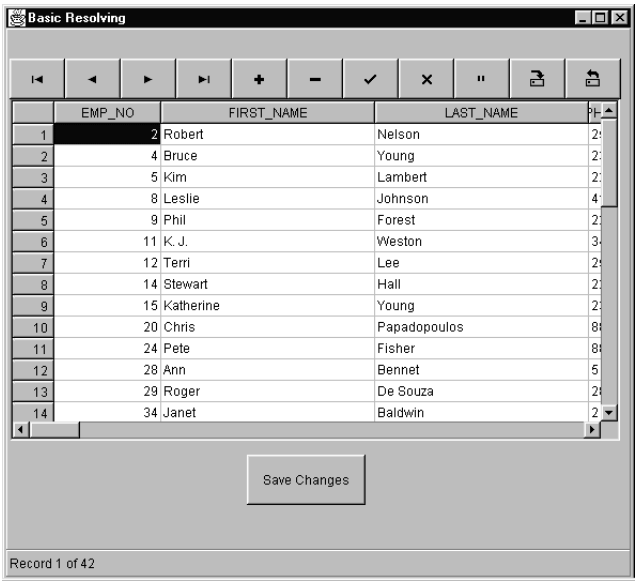
A *QueryDataSet* can be used in three different ways to fetch data.

- Unparameterized queries. In unparameterized queries, the query is executed and rows are fetched into the *QueryDataSet*.
- Parameterized queries. In a parameterized query, 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 5-23.
- Dynamic fetching of detail groups. When dynamic fetching of all detail groups is used, 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 7-2.

Tutorial: Querying a database using the JBuilder UI

The following tutorial shows how to provide data to an application using a *QueryDataSet* component. This example also demonstrates how to attach the resulting data set to a *GridControl* for data viewing and editing.

The finished example for this tutorial is available as a completed project in the samples\com\borland\samples\dx\QueryProvide directory of your JBuilder installation under the file name QueryProvide.jpr. The running application, including the addition of the information on resolving a query, looks like this:



Populating a data set

To create the application and populate a data set from a table,

1. Select File | Close. Select File | New. Double-click the Application icon and accept all defaults, except uncheck Use Only JDK & Swing Classes, to create a new application.
2. Select the Frame file in the Navigation pane. Select the Design tab to activate the UI Designer.
3. Click the *Database* component on the Data Express tab of the Component palette, then click in the Component tree to add the component to the application.

   Open the *connection* property editor for the *Database* component by double-clicking the *connection* property in the Inspector. Set the connection properties to the Local InterBase sample employee table, as follows:

Property name	Value
Connection URL	jdbc:odbc:DataSet Tutorial
Username	SYSDBA
Password	masterkey

The code generated by the designer for this step is:

```
database1.setConnection(new
 com.borland.dx.sql.dataset.ConnectionDescriptor
   ("jdbc:odbc:DataSet Tutorial", "SYSDBA", "masterkey", false,
    "sun.jdbc.odbc.JdbcOdbcDriver"));
```

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 gray area below the button. When the connection is successful, click OK.

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

4 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. Select the *query* property of the *QueryDataSet* component in the Inspector and set the following properties:

Property name	Value
Database	*database1*
SQL Statement	`select * from employee`
Place SQL text in resource bundle	unchecked

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

The code generated by this step is:

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

Creating the UI

This topic shows how to create a UI for your application using JBCL components. To create a UI using dbSwing components, see "Creating a database application UI using dbSwing components" on page 13-2.

To view the data in your application,

1 Add a *GridControl* component from the JBCL tab to the Designer. Set its *dataSet* property to *queryDataSet1*. You'll notice that the grid fills with data.

2 Add a *NavigatorControl* component from the JBCL tab to the Designer. Set its *dataSet* property to *queryDataSet1*. This will enable you to move quickly through the data set when the application is running, as well as provide a default mechanism for saving changes back to your data source.

3 Add a *StatusBar* control from the JBCL tab to the Designer. Set its *dataSet* property to *queryDataSet1*. Among other information, the status bar displays the current record.

4 Select Project | Properties. On the Run/Debug page, check Send Run Output To Execution Log. Click OK. Using the Execution Log instead of the Console Window enables you to view a list of activities as they occur, while running or debugging. You can also type notes into this window, and save the contents of the window to a text file. To open this window, choose View | Execution Log.

5 Select Run | Run to run the application and browse the data set.

To save changes back to the data source, you can use the Save Changes button on the navigator control or, for more control on how changes will be saved, see Chapter 6, "Saving changes back to your data source".

As a next step, you might want to edit and modify data, save changes back to your data source, filter data, sort data, or set various properties and methods of the resulting *QueryDataSet*.

Enhancing data set performance

This section provides some tips for fine-tuning the performance of *QueryDataSets* and *QueryProviders*. 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 metadata of a query" on page 5-18 for information on doing this.

- Set the *LoadOption* property on *Query/ProcedureDataSet*s 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 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.setCacheStatements(true)*. The prepared statements that are cached are not closed until one of the following happens:

  1 Some provider related property, like the *query* property, is changed.

  2 A *DataSet* component is garbage collected (statement closed in a *finalize()* method).

  3 *QueryDataSet.closeStatement()*, *ProcedureDataSet.closeStatement()*, *QueryProvider.closeStatement()*, or *ProcedureProvider.closeStatement()* are 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.

- For each call to *Database.saveChanges()*, set the *useTransactions* parameter to **false**. For each call to *Database.saveChanges()*, calls are made to disable/enable a JDBC driver's autocommit mode. If your application calls *database.saveChanges()* with the *useTransactions* parameter set to **false**, these calls will not be made and the transaction will not be committed.
- 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.

  If transactions are disabled, your application must call *database.commit()* or *connection.commit()*.

Persisting metadata of a query

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. The time overhead may, however, be removed from run time, and performed 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 MetaData" button in the Column Designer.

The query is now analyzed, and a set of property settings will be added to the code. To set the properties without using the Designer,

1. Set the StorageDataSet.metaUpdate property to MetaDataUpdate.NONE.
2. Set the StorageDataSet.tableName property to the table name for single table queries.
3. Set the Column.rowID property for the columns so that they uniquely and efficiently identify a row.
4. Change the query string to include a columns that is suitable to identify a row (see previous bullet), if not already included. Such columns should be marked invisible with the Column.visible property or the Column.hidden property.
5. Set the column properties Column.precision, Column.scale, and Column.searchable properties. These properties are not needed if the MetaDataUpdate property is set to render these at run time (see first step).

6 The Column.tableName property must be set for multi-table queries.

7 The Column.serverColumnName property must be 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.

If the dialog to get the username and password for a database hangs, open the data set you want to query in the jbInit() method, instead of letting it be opened implicitly when the controls bound to it open. Opening the data set in turn opens the connection, causing the password dialog to be displayed. You could also call the *database.openConnection()* method directly, since the password is a property of the connection.

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.

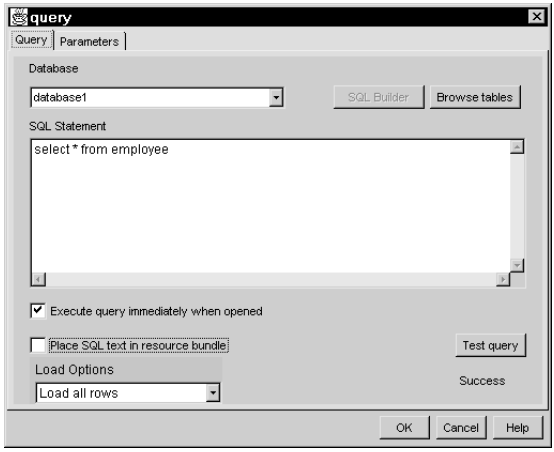
If a query is found to not be updateable and JBuilder cannot make it updateable by changing the query, the resulting data set will be read-only.

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 metadata of a query" on page 5-18 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.

Setting properties in the query dialog

The *query* property editor displays when you choose the *query* property of a *QueryDataSet*. This editor can be used to set the properties of the *QueryDescriptor* visually, but it also has several other uses. The dialog is displayed below. Each of the dialog options is explained in further detail as well.



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 Browse Tables 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 control.

- **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.

- 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.

- 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.

- When **Place SQL Text In Resource Bundle** is checked, upon exiting the *query* property editor, the Create ResourceBundle dialog box 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.

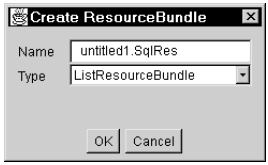
The Parameters page

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* object (or an instance of any of its subclasses), for example, a *DataSet* object. Select the *ReadWriteRow* object (or the *ReadWriteRow* subclass) that contains the values for your query parameters from the drop-down list. See "Using parameterized queries to obtain data from your database" on page 5-23 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,

- 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 issuing a right-click on 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_Messsage", "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
```

- Clicking the Cancel button (or unselecting 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 = getBundle("untitled3.SqlRes");
  public static final String getStringResource(String key) {
    return res.getString(key);
  }
  public Object[][] getContents() {
    return contents;
  }
}
```

If the SQL statement is changed, the changes would be 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.

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 runtime. A parameterized query uses these variables to replace literal data values, such as those used in a WHERE clause for comparisons,

that appear in a SQL statement. These variables are called *parameters*. Ordinarily, parameters stand in for data values passed to the statement. You provide the values for the parameters before running the query. By providing different sets of values and running the query for each set, you cause one query to return different data sets.

An understanding of how data is provided to a *DataSet* is essential to further understanding of parameterized queries, so read Chapter 3, "Understanding JBuilder database applications" and "Querying a database" on page 5-13 if you have not already done so. This topic is specific to parameterized queries.

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 runtime. When the values in the *ParameterRow* are changed, the grid will automatically refresh its display to reflect only the records that meet the criteria specified with the parameters.

A sample project ParamQuery.jpr, located in the samples\com\borland\samples\dx\ParamQuery directory of your JBuilder installation, contains the completed tutorial.

To create the application,

1. Create a new application.

   Select File | Close. Select File | New. Double-click Application to start the Application Wizard and create a new application. Accept all defaults, except uncheck Use Only JDK & Swing Classes.

2. Add a *Database* component and connect it to the Local InterBase sample employee table.

   To do this, select the Frame file in the Navigation pane. Select the Design tab in the UI Designer. Add a *Database* component from the Data Express tab to the Component tree. Open the *connection* property editor for the *Database* component by selecting the *connection* property ellipsis in the Inspector. Set the *connection* properties as follows:

Property name	Value
Connection URL	`jdbc:odbc:DataSet Tutorial`
Username	`SYSDBA`
Password	`masterkey`

   Click the Test Connection button to check that the connection properties have been correctly set. Results of the connection attempt are displayed in the gray area below the button. When the gray area beneath the button indicates `Success`, click OK to close the dialog.

3. Add a *ParameterRow* component from the Data Express tab to the Component tree.

   The following code is generated in the class definition:

   ```
   ParameterRow parameterRow1 = new ParameterRow();
   ```

Next, you will add two columns to the *ParameterRow*, low_no and high_no. After binding the *ParameterRow* to a *QueryDataSet*, a *TextField* components is used to change the value in the *ParameterRow* so that the query can be refreshed using these new values.

1 Select the *ParameterRow* in the Component tree. Double-click the component to display the column editor. Select <new column>, and set the following properties for the new column in the Inspector:

Property name	Value
columnName	low_no
dataType	*INT*
default	15

The following code is generated in the class definition:

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

The following code is generated in the *jbInit()* method:

```
column1.setColumnName("low_no");
column1.setDataType(com.borland.dx.dataset.Variant.INT);
column1.setDefault("15");
parameterRow1.setColumns(new Column[] {column1});
```

2 Select <new column> again to add the second column to the *ParameterRow*. In the Inspector, set the following properties for the new column:

Property name	Value
columnName	high_no
dataType	*INT*
default	50

The following code is generated in the class definition:

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

The following code is generated in *jbInit()* method:

```
column2.setColumnName("high_no");
column2.setDataType(com.borland.dx.dataset.Variant.INT);
column2.setDefault("50");
```

The column definition for the *parameterRow* object is modified as follows:

```
parameterRow1.setColumns(new Column[] {column1, column2});
```

3 Add a *QueryDataSet* component from the Data Express tab. Set the *query* property of the *QueryDataSet* component from the Inspector as follows:

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

On the Parameters page of the *query* dialog. Select *parameterRow1* in the drop-down list box to bind the data set to the *ParameterRow*.

On the Query page, click the Test Query button to ensure that the query is runnable. When the gray area beneath the button indicates `Success`, click OK to close the dialog.

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

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

To view and manipulate the data in your application,

1. Add a *GridControl* component from the JBCL tab to the UI Designer. Set its *dataSet* property to *queryDataSet1*. You'll notice that the grid fills with data.
2. Add a *NavigatorControl* component from the JBCL tab to the UI Designer. Set its *dataSet* property to *queryDataSet1*. When the application is running, the navigator will enable you to browse and edit the data in the table.

To add the controls to make the parameterized query variable at runtime,

1. Add a *TextField* component from the AWT tab to the UI Designer. This control holds the minimum value. Click the Events tab of the Inspector and triple-click on the *keyPressed* field. The Source pane will display and the cursor will be located at the correct spot for adding the following code:

```
//void textField1_keyPressed(KeyEvent e) {
   //when the Enter key is pressed
   if (e.getKeyCode() == KeyEvent.VK_ENTER) {
     try {
       //change the value in the parameter row and refresh the display
       parameterRow1.setInt("low_no", Integer.parseInt(textField1.getText()));
       queryDataSet1.refresh();
     }
     catch (Exception ex) {
       ex.printStackTrace();
     }
    }
// }
```

   The following code is automatically entered in the *jbInit()* method:

```
textField1.addKeyListener(new java.awt.event.KeyAdapter() {
```

   A new class with the same name is added to the Frame file as well.

   You can add a *LabelControl* to identify this field as the minimum field.

2. Select File | Save All to save your work so far.
3. Select the Design tab of the UI Designer. Add another *TextField* component from the AWT tab to the UI Designer. This control holds the maximum value. Select the Events tab of the Inspector and triple-click on the *keyPressed* field. The Source pane

will display and the cursor will be located at the correct spot for adding the following code:

```
//void textField2_keyPressed(KeyEvent e) {
   //when the Enter key is pressed
   if (e.getKeyCode() == KeyEvent.VK_ENTER) {
     try {
       //change the value in the parameter row and refresh the display
       parameterRow1.setInt("high_no", Integer.parseInt(textField2.getText()));
       queryDataSet1.refresh();
     }
     catch (Exception ex) {
       ex.printStackTrace();
     }
    }
 //}
```

The following code is automatically entered in the *jbInit()* method:

```
textField2.addKeyListener(new java.awt.event.KeyAdapter() {
```

A new class with the same name is added to the Frame file as well.

You can add a *LabelControl* to identify this field as the maximum field.

4 Select Run | Run to compile and run the application.

To test the example, enter a new value in the *TextField* that holds the minimum value, then press *Enter*. The grid displays only those values above the new minimum value. Enter a new value in the *TextField* that holds the maximum value, then press *Enter*. The grid displays only those values below the new maximum value.

To save changes back to the data source, click the Save Changes button on the Navigator.

Using parameters

To assign parameter values in a parameterized query, you must first create a *ParameterRow* and add named columns that will be the placeholders for 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 *addColumns* 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 *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 runtime by your application. The value in this field may come from a visual control or be generated programmatically. 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 runtime.

  In the "Tutorial: Parameterizing a query" on page 5-24, 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 runtime by your application. The value in this field may come from a visual control 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 runtime. See "Binding parameters" on page 5-29 for more information on binding parameters.

- 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 5-30.

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 control 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 control. To do this, your code would have to:

1. Obtain the value from the control 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 5-24 for an example of how to do this with JBuilder sample files.

Binding parameters

Binding parameters means allocating resources for the statement and its parameters both locally and on the server in order to improve execution of the query.

Binding parameters to a data set can be done by constructing a *QueryDescriptor* with a *ParameterRow* (containing values). In a parameterized query, parameters from the *ParameterRow* are used to set the parameters of the query. You can use the *getParameterRow* method on the *QueryDataSet* to obtain the current *ParameterRow*. You can add an *execute* method that takes a *ReadRow* to set the parameters.

To bind the values of a parameter to a data set,

1. Create a *ParameterRow* with column names explicitly defined for named parameters (such as :fld1, for example).
2. Get values for the columns (from a visual control, for example).
3. Put the values in the *ParameterRow*.
4. Execute or refresh the query (with *executeQuery()*, or *refresh()*, for example).

An example of binding parameters is included in "Tutorial: Parameterizing a query" on page 5-24.

Binding may also be done by invoking *refresh()* to re-execute the query with the new value in the parameter as discussed in "Re-executing the parameterized query with new parameters" on page 5-29.

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 7, "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. See "Binding parameters" on page 5-29 for more information on binding parameters.

Obtaining data through a stored procedure

In JBuilder, data is extracted from a server into a data set. This action is called "providing". Once the data is provided, you can view and work with the data locally in data-aware controls. You can store your data to local memory (*MemoryStore*) or to a local single-file database with a hierarchical directory structure (*DataStore*). When you want to save the changes back to your database, you must resolve the data. This process is discussed in more detail in "Understanding JBuilder's DataExpress architecture" on page 3-3.

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, you need a *Database* component, a *ProcedureDataSet* component, and a *ProcedureDescriptor*. You can provide this information programmatically, or by using JBuilder design tools.

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

When providing data from JDBC data sources, the *ProcedureDataSet* has built-in functionality to fetch data from a stored procedure that returns a cursor to a result

set. The following properties of the *ProcedureDescriptor* object affect the execution of stored procedures:

Property	Purpose
database	Specifies what *Database* connection object to run the procedure against.
procedure	A Java String representation of a stored procedure escape sequence or SQL statement that causes a stored procedure to be executed.
parameters	An optional *ReadWriteRow* from which to fill in parameters. These values can be acquired from any *DataSet* or *ReadWriteRow*.
executeOnOpen	Causes the *ProcedureDataSet* 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**.
loadOption	An optional integer value that defines 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 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 5-37 provides an example.

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

Tutorial: Accessing data through a stored procedure

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 *GridControl* and a *NavigatorControl* for data viewing and editing.

The finished example for this tutorial may be available as a completed project in the samples\com\borland\samples\dx\StorProc directory of your JBuilder installation under the file name StorProc.jpr or check http://www.borland.com/techpubs/jbuilder/ for the latest updates. Other sample applications referencing stored procedures on a variety of servers are available in the samples\com\borland\samples\dx\StoredProcedure directory, and a sample of providers is available in the samples\com\borland\samples\dx\providers 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 local InterBase server (in the directory set up in "Installing Local InterBase Server" on page 2-3). This procedure is written in the InterBase language. These procedures will be used both in this section and in the "Tutorial: Saving changes with a NavigatorControl" on page 6-5 and "Tutorial: Saving changes with a ProcedureResolver" on page 6-8.

1. Select File | Close from the menu.
2. Select File | Open and open the project ProcSetUp.jpr, which may be located in the samples\com\borland\samples\dx\StorProc\ProcSetUp 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" at the end of this topic.
3. Right-click *createprocedures.java* in the Navigation pane, select Run. This step creates the tables and procedures on the server.
4. Select File | Close from the menu.

Adding the DataSet components

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

1. Select File | Close.
2. Select File | New and double-click the Application icon. Accept all defaults, except uncheck Use Only JDK & Swing Classes.
3. Select Frame1.java in the Navigation pane. Select the Design tab to activate the UI Designer.
4. Place a *Database* component from the Data Express tab of the Component Palette on the Component tree.
5. Open the *connection* property editor for the *Database* component by selecting the *connection* property ellipsis in the Inspector. Set the *connection* properties to the Local InterBase sample tables by setting the properties as indicated in the following table. These steps assume you have completed "Installing Local InterBase Server" on page 2-3.

Property name	Value
Connection URL	`jdbc:odbc:DataSet Tutorial`
Username	`SYSDBA`
Password	`masterkey`

The code generated by the designer for this step is:

```
database1.setConnection(new com.borland.dx.sql.dataset.ConnectionDescriptor
   ("jdbc:odbc:DataSet Tutorial", "SYSDBA", "masterkey", false,
    "sun.jdbc.odbc.JdbcOdbcDriver"));
```

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 gray area below the button. When the text indicates `Success`, click OK to close the dialog.

6 Place a *ProcedureDataSet* component from the Data Express tab of the Component Palette on the Component tree. In the Inspector, set the *procedure* property of the *ProcedureDataSet* component from the Inspector 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 6-8.

The code generated by this step is:

```
procedureDataSet1.setProcedure (new
  com.borland.dx.sql.dataset.ProcedureDescriptor(database1, "select * from
  GET_COUNTRIES", null, true, Load.ALL));
```

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 5-34 for more information.

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

Adding visual controls

This topic shows how to create a UI for your application using JBCL components. To create a UI using dbSwing components, see "Creating a database application UI using dbSwing components" on page 13-2.

To view the data in your application,

1 Place a *GridControl* component from the JBCL tab of the Component Palette on the UI Designer. In the Inspector, set its *dataSet* property to *procedureDataSet1*. You'll notice that the grid fills with data.

2 Select Run | Run to compile the application, and to browse the data set.

You must add resolving capability to your application in order to edit, insert, and delete data in the running application. See:

- "Tutorial: Saving changes with a NavigatorControl" on page 6-5
- "Tutorial: Saving changes with a ProcedureResolver" on page 6-8

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.

  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 5-37. For an example using InterBase stored procedures, see "Example: Using InterBase stored procedures" on page 5-36.

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 StorProc 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 from the menu.
2. Select File | New and select the Class icon.

3 Change the file directory and project name to StorProc\ProcSetUp\ProcSetUp.jpr in the Project Wizard.

4 Change the Class Name to ProcSetUp in the New Java File dialog. Click OK to create the file ProcSetUp.java.

5 Make sure that the file ProcSetUp.java is selected in the Navigation pane and that the Source window is selected in the UI Designer. 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 database1 = new Database();
    database1.setConnection(new ConnectionDescriptor("jdbc:odbc:dataset tutorial",
      "sysdba","masterkey", false, "sun.jdbc.odbc.JdbcOdbcDriver"));
    try { database1.executeStatement("DROP PROCEDURE GET_COUNTRIES"); }
      catch (Exception ex) {};
    try { database1.executeStatement("DROP PROCEDURE UPDATE_COUNTRY"); }
      catch (Exception ex) {};
    try { database1.executeStatement("DROP PROCEDURE INSERT_COUNTRY"); }
      catch (Exception ex) {};
    try { database1.executeStatement("DROP PROCEDURE DELETE_COUNTRY"); }
      catch (Exception ex) {};
    database1.executeStatement(getCountriesProc);
    database1.executeStatement(updateProc);
    database1.executeStatement(deleteProc);
    database1.executeStatement(insertProc);
    database1.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;";
}
```

6 Select *createprocedures.java* in the Navigation pane, then select Run | Run from the menu. This step creates the tables and procedures on the server.

7 Select File | Close from the menu.

This is a very simple procedure.

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 Statement field in the *procedure* property editor, or in code:

```
select * from ORG_CHART
```

For a look at more complicated InterBase stored procedures, use JDBC Explorer 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" on page 5-32 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. Define the *ProcedureDescriptor* to call this function from JBuilder.
   1. Select the Frame file in the Navigation pane, then select the Design tab.
   2. Place a *ProcedureDataSet* from the Data Express tab to the Component Tree.
   3. Select the *procedure* property to bring up the *ProcedureDescriptor* dialog.
   4. Select *database1* from the *Database* drop-down list.
   5. Enter the following escape syntax in the Stored Procedure Escape or SQL Statement field, or in code:

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

   6. 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/com/borland/samples/dx/storedProcedure 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

As shown in Figure 5.1, "DataExpress Architecture," on page 5-3, 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\com\borland\samples\dx\providers directory of your JBuilder installation. The sample file TestFrame.java is an application with a frame that contains a *GridControl* and a *NavigatorControl*. Both visual controls 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 undelimited 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. It also determines what kind of metadata discovery is performed when executing a query or a stored procedure against a SQL server database. 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* component 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* component 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 controls.

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*.

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*. The following topics are discussed in the following sections:

- Metadata and how it is obtained
- Non-metadata Column properties
- Setting Column properties
- Persistent columns
- Combining live metadata with persistent columns
- Controlling column order in a DataSet

Metadata and how it is obtained

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*. When you obtain data from a data source (a process referred to as *providing*), 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, it typically 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. A query may be re-executed if *rowId*s need to be added later. The *QueryDataSet* component then creates *Column* objects automatically as needed at run time. One *Column* is created for each column in the provided data. Each *Column* then gets some of its properties from the metadata, such as *columnName*, *tableName*, *rowId*, *searchable*, *precision*, *scale*, and so on.

You can suppress the initial metadata query to increase performance if your program already knows what the necessary metadata is. It is safe to suppress the initial metadata query only if you are certain that the schema (structure) of your database will not change in ways that affect the critical metadata needed by your *DataSet*, such as *rowId* and *tableName*. If the schema does change later, your program will most likely fail to run properly.

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 controls, 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

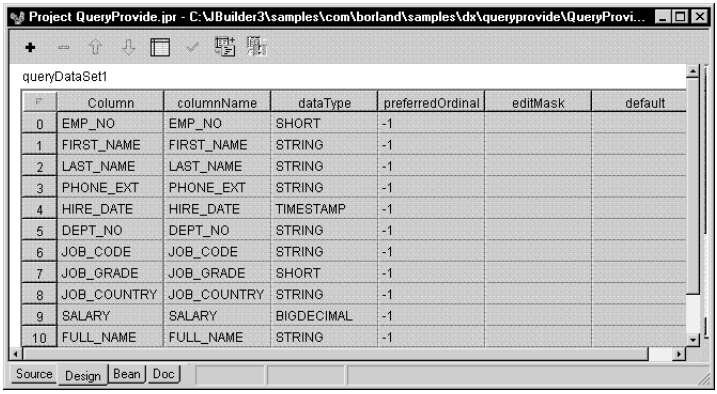
One way to view column properties information is by using the Column Designer. The Column Designer displays information for selected properties, such as the data type for the column, in a navigable grid. Changing, or setting, a property in the Column Designer makes a column persistent. The column properties can be modified in the Column Designer or in the Inspector. You can change which properties display in the Column Designer by right-clicking on a header, and selecting a different property to display.

To display the Column Designer,

1. Open any project that includes a *DataSet* object. In this example, select samples\com\borland\samples\dx\queryprovide\queryprovide.jpr from your JBuilder installation. (This directory is available only if you selected the Typical option from the JBuilder setup program, or selected Sample files from the Custom option.)

2 Select the file QueryProvideFrame.java file and click the Design tab from the bottom of the right pane of the AppBrowser.

3 Double-click the *queryDataSet1* object from the Component tree. This displays the Column Designer for the data set in the Design window. The Column Designer looks like this for the Local InterBase EMPLOYEE sample table:



	Column	columnName	dataType	preferredOrdinal	editMask	default
0	EMP_NO	EMP_NO	SHORT	-1		
1	FIRST_NAME	FIRST_NAME	STRING	-1		
2	LAST_NAME	LAST_NAME	STRING	-1		
3	PHONE_EXT	PHONE_EXT	STRING	-1		
4	HIRE_DATE	HIRE_DATE	TIMESTAMP	-1		
5	DEPT_NO	DEPT_NO	STRING	-1		
6	JOB_CODE	JOB_CODE	STRING	-1		
7	JOB_GRADE	JOB_GRADE	SHORT	-1		
8	JOB_COUNTRY	JOB_COUNTRY	STRING	-1		
9	SALARY	SALARY	BIGDECIMAL	-1		
10	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 Right-click on a header, 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, double-click on any UI component in the Component tree, or single-click on 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 "Making columns persistent" on page 12-21 for more information on using the Column Designer.

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 runtime.

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*.

Prior to the implementation of this feature, you could set the various properties necessary to persist the metadata. The runtime code of DataExpress does this automatically, as well, however, some JDBC drivers are slow at responding to metadata inquiries. With JBuilder setting this up at design time, and generating the necessary code for runtime, performance will be improved.

See also "Persisting metadata of a query" on page 5-18

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 code added by the developer. However, other properties, like *editMask* will not be touched.

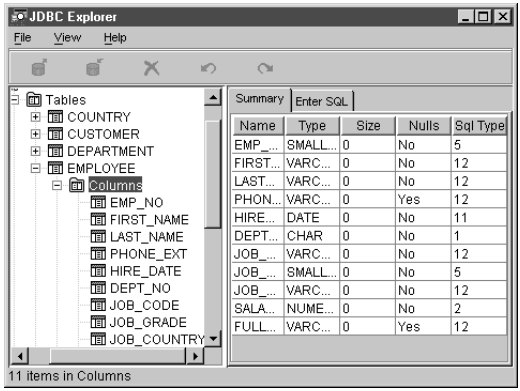
Prior to the implementation of this feature, you could set the various properties necessary to persist the metadata. The runtime code of DataExpress does this automatically, as well, however, some JDBC drivers are slow at responding to metadata inquiries. With JBuilder setting this up at design time, and generating the necessary code for runtime, performance will be improved.

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 JDBC Explorer

The JDBC Explorer is an all-Java, 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.

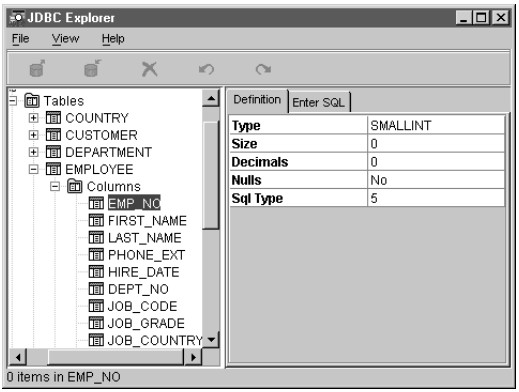
To display the JDBC Explorer, select Tools | JDBC Explorer from the JBuilder menu.



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 JDBC Explorer, 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 5-14.

  Tip You can click the Test Query button on the *query* property editor to test your query and its connection to the *Database*.

  2. Open the UI Designer by selecting the *Frame* container object in the Navigation pane, and then clicking the Design tab in the AppBrowser.
  3. In the Component tree, select the *StorageDataSet* component.
  4. Double-click 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 5-14 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* component 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* components for every column found in the data provided by the data source. It discards any *Column* components that were explicitly added previously, or automatically created for a previous batch of data. This discarding of previous *Column* components 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 "Specifying required data in your application" on page 12-21. 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 plus (+) sign in the Component tree, 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 Component tree. 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 Component tree 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 Component tree. 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 controls, 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 about creating columns explicitly in code in the section "Persistent columns" on page 5-46.

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

1. Follow the first 3 steps in "Setting column properties" on page 5-45 to obtain the metadata into the columns listed in the Component tree. (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.

Chapter

6

Saving changes back to your data source

After data has been provided to your application, you can make changes to the local subset of the data in the *StorageDataSet*. 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.

When working on a local subset of server data, you are working in isolation from the data source. On the client machine, the data can be stored locally in either a *DataStore* (see the *DataStore Developer's Guide*) or in memory in a *MemoryStore*.

Between the time that the local subset is provided, 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 as errors can occur while saving changes, such as violation of server integrity constraints and resolution conflicts. A resolution conflict can happen from deleting a row that is already 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 6-2 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 7-8.
- "Saving changes back to your data source with a stored procedure" on page 6-5 covers resolving changes made to a *ProcedureDataSet* back to its data source.
- "Resolving data from multiple tables" on page 6-10 provides the necessary settings for resolving changes when a query involves more than one table.
- "Streaming data" on page 6-12 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 6-14 describes how to set custom resolution rules using the *QueryResolver* component and resolver events.
- "Exporting data" on page 8-5 describes how to export data to a text file.

Saving changes from a QueryDataSet

This topic explores the basic resolver functionality provided by the DataExpress package. It extends the concepts explored in "Querying a database" on page 5-13 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 for this tutorial. These files are located in the samples\com\borland\samples\dx\QueryProvide directory under the file name QueryProvide.jpr.

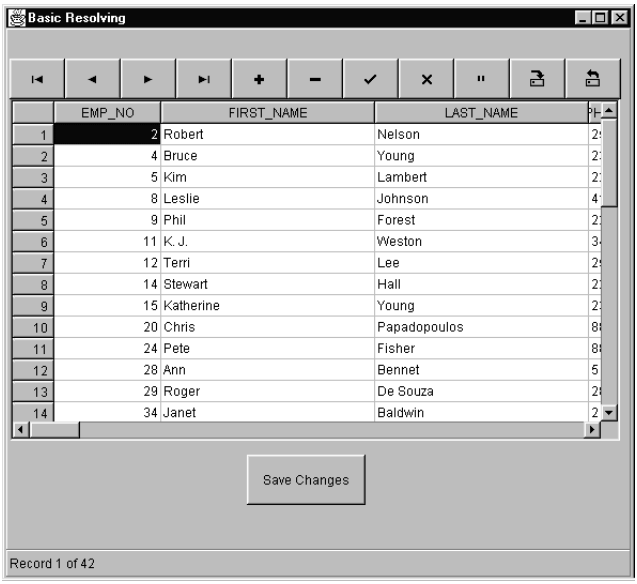
The "Querying a database" tutorial explored the providing phase where data is obtained from a data source. The tutorial instantiated a *QueryDataSet* and associated components, and displayed the data stored in the Local InterBase Server employee sample file in a grid. This topic expands that example by

- Adding basic resolving capability in three ways:

  1 A button that contains basic resolving code.
  2 A *NavigatorControl* whose Save button also performs a basic resolving function.
  3 A *JdbnavToolBar* whose Save Button also performs a basic resolving function.

  When either the custom button or the navigator's Save button is pressed, the changes made to the data in the *QueryDataSet* are saved to the employee data file using the DataExpress package's default query resolver.
- Adding a *StatusBar* or *JdbStatusLabel* control and connecting it to the same *QueryDataSet* so it can display messages generated by the *QueryDataSet*.

If you prefer to review the source code for the completed tutorial, the completed project files are located in the samples\com\borland\samples\dx\BasicResolve directory of your JBuilder installation under the file name BasicResolve.jpr. 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\com\borland\samples\dx\QueryProvide 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 Navigation pane. Select the Design tab in the Content pane.
3. Add a *NavigatorControl* component from the JBCL tab of the Component palette. Place it above the grid. You may want to resize the grid to make room for the navigator, or enlarge the panel and move components around to make room for it.

   In the Inspector, set the *dataSet* property of the *NavigatorControl* to the instantiated data set, *queryDataSet1*.
4. Add a *StatusBar* control from the JBCL tab of the Component palette. Place it at the bottom of the application UI.

   In the Inspector, set the *dataSet* property of the *StatusBar* to the instantiated data set, *queryDataSet1*.

5 Add a *ButtonControl* component from the JBCL tab of the Component palette. Set the button's *label* property to `Save Changes`. (See the finished application at the beginning of this tutorial for general placement of the controls in the UI.)

6 Select the *ButtonControl*, then click the Events tab of the Inspector. 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 StatusBar if the application includes one,
// or displays an error dialog if there isn't
    DataSetException.handleException(ex);  }
```

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

7 Run the application by selecting Run | Run. The application compiles and displays in a separate window. Data is displayed in a grid, with a Save Changes button, the navigator, and a status bar 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 navigator to scroll through the data displayed in the grid. The status bar 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 grid by inserting, deleting, and updating data. You can save the changes back to the Local InterBase employee database choosing either,

- the Save Changes button you created, or
- the Save button of the *NavigatorControl*

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

until you become more familiar with the constraints on this table. For more information, see "Tips on using InterBase" on page 2-5.

For a list of more advanced resolving topics, see "Saving changes back to your data source with a stored procedure" on page 6-5.

Saving changes back to your data source with a stored procedure

You can add a resolving phase where you save changes back to your data source using either the basic resolving capability in a *NavigatorControl* or by setting properties of a *ProcedureResolver* to specify special coding of the stored procedure on the database on which the data should be resolved.

This topic explores the basic resolver functionality provided by the DataExpress package for *ProcedureDataSet* components. It extends the concepts explored in "Obtaining data through a stored procedure" on page 5-30, and adds a resolving phase where you save your changes back to the data source.

The "Tutorial: Accessing data through a stored procedure" on page 5-31 explored the providing phase where data is obtained from a data source with a stored procedure. The tutorial instantiates a table and insert, update, and delete procedures on the server. Then, it instantiates a *ProcedureDataSet* component and associated components, and displays the data returned from the Local InterBase Server procedure in a grid. The following tutorial expands the providing phase 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 contains basic resolving code or a *NavigatorControl* whose Save button also performs a basic resolve function. See "Tutorial: Saving changes with a NavigatorControl" on page 6-5.
- A *ProcedureResolver* that requires special coding of the stored procedure on the database on which the data should be resolved. See "Coding stored procedures to handle data resolution" on page 6-7. An example of this is available in "Tutorial: Saving changes with a ProcedureResolver" on page 6-8.
- Also, see "Example: Using InterBase stored procedures with return parameters" on page 6-10 for information on that topic.

Tutorial: Saving changes with a NavigatorControl

The following tutorial shows how to save changes to your database using JBuilder's UI Designer, a *NavigatorControl*, and a *ProcedureDataSet* component.

The finished example for this tutorial may be available as a completed project in the samples\com\borland\samples\dx\StorProc directory of your JBuilder installation under the file name StorProc.jpr. Other sample applications referencing stored procedures on a variety of servers are available in the samples\com\borland\samples\dx\StoredProcedure directory.

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

1. Select File | Close. Select File | Open. Open the project file you created for "Tutorial: Accessing data through a stored procedure" on page 5-31. We will add resolving capability to the existing project.
2. Select Frame1.java in the Navigation pane. Select the Design tab to activate the UI Designer.
3. Place a *NavigatorControl* component from the JBCL tab of the Component Palette on to the UI Designer below the grid. The *NavigatorControl* has buttons that allow you to move around the grid, and negate or save changes to your data. You may need to resize the grid to make room for the navigator, or enlarge the panel and move components around to make room for it.
4. Set the *dataSet* property of the *NavigatorControl* to *procedureDataSet1* by selecting the *NavigatorControl*, selecting the *dataSet* property in the Inspector, then selecting *procedureDataSet1* from the list. This connects the *NavigatorControl* to the data set. You can set the *buttonType* property to *TextOnly* if you are not familiar with the navigator symbols.
5. Place a *StatusBar* control from the JBCL tab of the Component Palette on the UI Designer below the *NavigatorControl*. This control displays the current record and any other pertinent information related to the record. Set the *dataSet* property of the *StatusBar* to *procedureDataSet1*. This connects the *StatusBar* to the data set.

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 6-8). Then proceed with the following steps:

1. Set the *rowID* property of the key column to **True**. To do this, click the + sign to the left of *procedureDataSet1* in the Component Tree to expose the columns of the data set. Select the key column named COUNTRY. In the Inspector, select the *rowID* property and select **True** from the list.
2. Verify that the *resolvable* property of the *procedureDataSet1* is set to **True**.
3. Select Run | Run to run the application.

The application compiles and displays in a separate window. Data is displayed in a grid with the navigator and a status bar 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 navigator to scroll through the data displayed in the grid. The status bar 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 *ButtonControl* coded to handle saving changes in place of the *NavigatorControl*. 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 StatusBar if the application includes one,
// or displays an error dialog if there isn't
   DataSetException.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 6-8. In the case of InterBase, also see "Example: Using InterBase stored procedures with return parameters" on page 6-10.

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\com\borland\samples\dx\StoredProcedure directory.

To complete the application and save changes back to the COUNTRY table with custom defined insert, update, and delete procedures,

1. Select File | Close from the menu. Select File | Open. Open the project file you created for "Tutorial: Accessing data through a stored procedure" on page 5-31. Resolving capability will be added to the existing project.
2. Place a *NavigatorControl* component from the JBCL tab of the Component Palette on to the UI Designer below the grid. The *NavigatorControl* has buttons that allow you to move around the grid, and negate or save changes to your data. You may need to resize the grid to make room for the navigator, or enlarge the panel and move components around to make room for it.
3. Set the *dataSet* property of the *NavigatorControl* to *procedureDataSet1* by selecting the *NavigatorControl*, selecting the *dataSet* property in the Inspector, then selecting *procedureDataSet1* from the list. This will connect the *NavigatorControl* to the data set. You can set the *buttonType* property to *TextOnly* if you are not familiar with the navigator symbols.

   In addition to moving around the grid, a navigator provides a Save Changes button. At this point, this button will not do anything. Once we provide a custom resolver via a *ProcedureResolver*, the Save Changes button will call the insert, update, and delete procedures. You could instead use a *ButtonControl* as indicated in the "Tutorial: Saving changes with a NavigatorControl" on page 6-5.
4. Place a *StatusBar* control from the JBCL tab of the Component Palette on the UI Designer below the *NavigatorControl*. This control will display the current record and any other pertinent information related to the record. Set the *dataSet* property of the *StatusBar* to *procedureDataSet1*. This will bind the *StatusBar* to the data set.

At this point in the tutorial, 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.

1. Place a *ProcedureResolver* component from the Data Express tab of the Component Palette on the Component tree. Set the *database* property of the *ProcedureResolver* to the instantiated database, *database1*.

2. Set the *deleteProcedure* property of the *ProcedureResolver* to DELETE_COUNTRY. To do this, double-click in the *deleteProcedure* property to bring up the *deleteProcedure* dialog.
   1. Set the *Database* property to *database1*.
   2. 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
      ```

   3. Edit this statement to:

      ```
      execute procedure DELETE_COUNTRY :COUNTRY
      ```

      See the text of the procedure in "Creating tables and procedures for the tutorial manually" on page 5-34 or through the JDBC Explorer.
   4. Click Test Procedure. When the area beneath the button indicates Success, click OK.
3. Set the *insertProcedure* property to INSERT_COUNTRY. To do this, double-click in the *insertProcedure* property of the *ProcedureResolver*.
   1. Set the *Database* field to *database1*.
   2. Click Browse Procedures, then double-click the procedure named INSERT_COUNTRY.
   3. Edit the generated code to read:

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

   4. Click Test Procedure. When the area beneath the button indicates Success, click OK.
4. Set the *updateProcedure* property to UPDATE_COUNTRY. To do this, double-click in the *updateProcedure* property of the *ProcedureResolver*.
   1. Set the *Database* property to *database1*.
   2. Click Browse Procedures, then double-click the procedure named UPDATE_COUNTRY.
   3. Edit the generated code to read:

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

      Click Test Procedure. When the area beneath the button indicates Success, click OK.
5. Select *procedureDataSet1* in the Component tree. Set the *resolver* property to *procedureResolver1*.
6. Select Run | Run to run the application.

When you run the application, you can browse, edit, insert, and delete data in the grid. Save any change you make with the Save Changes button on the navigator. 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 following function:

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

Drivers:

- Visigenics 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*
- *DataSet - 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 is 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 *QueryDataSet*s unless the *metaDataUpdate* property does not include *metaDataUpdate.TABLENAME*.

What if a table is not updatable?

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

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

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.

Streaming data

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. See the topic "Creating a distributed database application using DataSetData" on page 14-1 for an example using *DataSetData* and RMI.

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

Example: Using streamable data sets

One example of when you would use a streamable *DataSet* is in a 3-tier system with a Java server application that responds to client requests for data from certain data sources. The server may use JBuilder *QueryDataSet*s or *ProcedureDataSet*s 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

As shown in the DataExpress diagram in Chapter 3, "Understanding JBuilder database applications", JBuilder makes it easy to write a custom resolver 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\com\borland\samples\dx\providers directory of your JBuilder installation (Enterprise version only). The sample file TestApp.java is an application with a frame that contains a *GridControl* and a *NavigatorControl*. Both visual controls 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 undelimited text file. The structure of data.txt is described in the interface file DataLayout.java.

An example of writing a custom *ProcedureResolver* is available in the "Tutorial: Saving changes with a ProcedureResolver" on page 6-8.

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 5-13 for how to do this.
2. Select the *Frame* class and click the Design tab to display the UI Designer.
3. Click the *QueryResolver* component (shown here) from the Data Express tab of the Component Palette.

4. Click (anywhere) in the Component tree or the UI Designer to add it to your application.

The UI Designer generates source code that creates a default *QueryResolver* object (named *queryResolver1* by default) that looks like this:

```
queryDataSet1.setResolver(queryResolver1);
```

To connect the *QueryResolver* to the appropriate *DataSet*,

1. Select the Frame file in the Navigation pane.
2. Select the Design tab.
3. Select the *DataSet* component in the Component tree.
4. Set the *resolver* property of the *DataSet* to the appropriate *QueryResolver*, for example, *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 *DataSet*.

Intercepting resolver events

You control the resolution process by intercepting *QueryResolver* events. When the *QueryResolver* object is selected in the Component tree, 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 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*.

Tutorial: Using resolver events

For an example of resolver events, see ResolverEvents.jpr and associated files in the samples\com\borland\samples\dx\ResolverEvents directory of your JBuilder installation. In the ResolverEvents application,

- A grid is bound to the Customer table in the Local InterBase Server sample database
- The Save Changes button creates a custom *QueryResolver* object which takes control of the resolution process.

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

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

Writing a custom data resolver

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

In order to resolve data changes back to a source,

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

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

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

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

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

It is important to note that

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

3 Close each of the *DataSetView*s after the data has been resolved, or if an exception occurs during resolution. If the *DataSetView*s 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 *DataSet*s 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 7-8.

Chapter

7

Establishing a master-detail relationship

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

In JBuilder, you can join, or link, two data sets that have at least one common field with 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. To resolve data from multiple data sets, refer to “Resolving data from multiple tables” on page 6-10.

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.

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 6-10.

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.

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 5-30.

When the master data set has two or more detail data sets associated with it, and the *fetchAsNeeded* property of each is **true**, the details remember what detail groups they have attempted to fetch via a query or stored procedure that is parameterized on the active master row linking columns. This memory can be cleared by calling the *StorageDataSet.empty()* method. There is no memory for *masterLink* properties that do not set *fetchAsNeeded* to **true**.

When the detail data set is a *TableDataSet*, the *fetchAsNeeded* parameter is ignored and all data is fetched at once.

Editing data in master-detail data sets

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 *GridControl*, 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. Skip this step if you currently have an application set up with at least two data set components, one of which represents the master data set and another which represents the detail data set.

   Create 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 the tutorial that uses the sample database files shipped with JBuilder, "Tutorial: Creating a master-detail relationship" on page 7-5.

2. Select the detail data set and select the *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 the previous section for more discussion on this option.
- To verify that the data sets are properly connected, click Test Link. The gray status area beneath the button will indicate `Running`, `Success`, or `Failed`.
- To complete the specification, click OK.

3 Add two visual controls (such as grids) 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.

4 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 7-8.

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 employee.gdb, and one that selects all of the employees. This tutorial is available as a finished project in the samples\com\borland\samples\dx\MasterDetail directory of your JBuilder installation under the project name MasterDetail.jpr.

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 grids and as you navigate through the COUNTRY grid, the EMPLOYEE grid displays all of the employees who live in the country indicated as the current record.

To create this application,

1. Select File | Close All. Select File | New, and double-click the Application icon. Accept all defaults, except uncheck Use Only JDK & Swing Classes.
2. Select Frame1.java in the Structure pane and then select the Design tab of the Content pane. Add a *Database* component from the Data Express tab to the Component tree. Set the *connection* property for the *Database* component as follows, assuming your system is set up as in Chapter 2, "Installing and setting up JBuilder for database applications":

Property name	Value
Connection URL	`jdbc:odbc:DataSet Tutorial`
Username	`SYSDBA`
Password	`masterkey`

   Click the Test Connection button to check that the *connection* properties have been correctly set. Results of the connection attempt are displayed in the gray area below the button. When the gray area beneath the button indicates `Success`, click OK to close the dialog.
3. Add a *QueryDataSet* component from the Data Express tab to set up the query for the master data set. Set the *query* properties of the *QueryDataSet* component from the Inspector as follows:

Property name	Value
Database	*database1*
SQL Statement	`select * from COUNTRY`

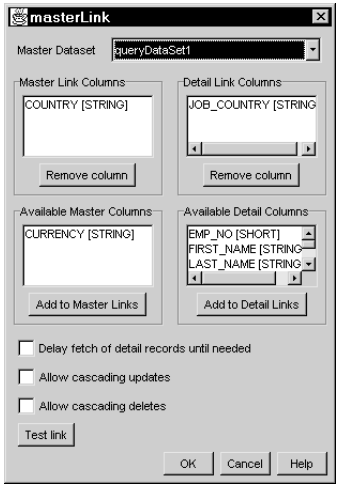
   Click the Test Query button to ensure that the query is runnable. When the gray area beneath the button indicates `Success`, click OK to close the dialog.
4. Add another *QueryDataSet* component from the Data Express tab to the Component tree. Select the *query* property. This will set up the query for 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 gray area beneath the button indicates `Success`, click OK to close the dialog.
5. Select the *masterLink* property for the detail data set (*queryDataSet2*). In the *masterLink* custom 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 7-2.
- Click Test Link. When successful, click OK.

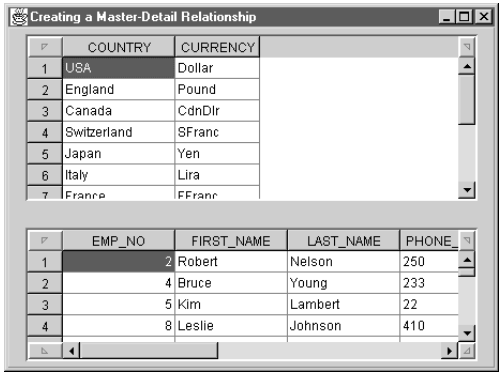
The dialog should look like this when you are done:



6 Add two *GridControl* components from the JBCL tab to the UI Designer. Bind the data sets to the grids by setting the first *GridControl*'s *dataSet* property to *queryDataSet1* and setting the second *GridControl*'s *dataSet* property to *queryDataSet2*.

7 Compile and run the application by selecting Run | Run.

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 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 *ButtonControl* 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 "Saving changes from a QueryDataSet" on page 6-2.

  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 7-9 for more information.

- Place a *QueryResolver* in your application to customize any resolution. See "Customizing the default resolver logic" on page 6-14 for more information.

- Place one or more *NavigatorControl*s in your application and use the Save button to save changes.

  If both data sets are *QueryDataSets*, you can use a separate *NavigatorControl* for each data set, or use a single *JdbNavToolBar* for both data sets. The JdbNavToolBar component, unlike its JBCL counterpart, automatically attaches itself to *DataSet*s (whichever *DataSet* has focus), so you do not need to set its *dataSet* property.

Warning When two *NavigatorControl*s are used, its Save button will only save changes to the data set it is bound to with the *NavigatorControl*'s *dataSet* property. It is possible to save a master without saving the details, or vice versa, with this system.

See also Chapter 6, "Saving changes back to your data source"

Resolving master-detail data sets to a JDBC data source

Because a master-detail relationship by definition includes at least two sets of data, the simplest way to resolve data back to the data source is to use the *saveChanges(DataSet[])* method of the *Database* component (assuming that *QueryDataSet*s 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 *NavigatorControl* 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.

Chapter

8

Importing and exporting data from a text file

In JBuilder, a *TableDataSet* component is used to store data you import from a text file. Once the data is provided to the data set, you can view and work with the data in JBuilder. To save changes back to the text file, you export the data back to the text file (see "Exporting data" on page 8-5).

To import data from a text file, you 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. You create *Column* objects so the *TableDataSet* set knows the type of data and the name of the field for each column of data.

You define the columns of a *TableDataSet* 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 "An introductory database tutorial using a text file" on page 5-3. Your text file has a valid .SCHEMA file only if it has previously been exported by JBuilder.

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 FrameNoSchema.jpr in the samples\com\borland\samples\dx\FrameNoSchema directory of your JBuilder installation.

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

1. Open a text editor, such as Notepad.
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* 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.

To read in 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, except uncheck Use Only JDK & Swing Classes.
2. Select the Frame file in the Navigation pane. Select the Design tab from the Content pane.
3. Add a *TextDataFile* component from the Data Express tab of the Component palette to the Component tree or to the Design window. Select the component and set the following properties in the Inspector:

Property name	Value
delimiter	" (double quote)
separator	, (comma)
fileName	`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.

   You can use the Browse button to specify the complete path and file name for the *fileName* field.

4. Add a *TableDataSet* component from the Data Express tab of the Component palette to the Component tree or to the Design window. Select the component and set its *dataFile* property 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 8-3. 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 "+" symbol 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 visual control, such as a *GridControl*, from the JBCL tab of the Component palette to the UI Designer.

This topic shows how to create a UI for your application using JBCL components. To create a UI using dbSwing components, see "Creating a database application UI using dbSwing components" on page 13-2.

2 In the Inspector, set the *dataSet* property of the visual control 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 control to view data, you must explicitly open the file in the source code to have access to the data.

3 Select Run | Run to compile and run the application.

When you run this application, the data in the text file is loaded into a *TableDataSet* and displayed in the visual grid control 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 8-5 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 Design window and with code in the Source window. Adding columns in the UI Designer is covered in "Tutorial: Importing data from a text file" on page 8-1. 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 Source window, follow these steps.

1 Open the Source window.

2 Define new *Column* objects in the class definition for Frame1. Select Frame1.java in the Structure pane, then select th 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();
```

3 Find the *jbInit()* method in the Source window. 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);
```

4 Add the new columns to the *TableDataSet* in the same source window and same *jbInit()* method, as follows:

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

5 Compile the application to bind the new *Column* objects to the data set, then add any visual controls.

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*. *exportDisplayMask* is used for import if there is no .SCHEMA 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 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 8-1.

```
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 8-9.

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 from a TableDataSet 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 8-1, 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 mentioned above. If you have created this project, open it now (use File | Open).
2. Select the Frame file in the Navigation pane. Select the Design tab of the Content pane.
3. Add a *ButtonControl* component from the JBCL tab to the UI Designer. With the *ButtonControl* selected, click the Properties tab of the Inspector. Set the text of the *label* property to `Save Changes`.
4. With the *ButtonControl* selected, click the Events tab of the Inspector. 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 {
     tableDataSet1.getDataFile().save(tableDataSet1);
     System.out.println("Changes saved");
 }
 catch (Exception ex) {
   System.out.println("Changes NOT saved");
   System.err.println("Exception: " + ex);
 }
```

5. Run the application.

When you run the application, if it compiles successfully, the application appears in its own window. Data is displayed in a grid, 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, such as Notepad. 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 = 8859_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 from a TableDataSet to a text file" on page 8-5, 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 in the Structure pane. Expand the *TableDataSet* component in the tree by clicking on the + icon to its left, select *<new column>*, then modify the column's properties as follows:
   - *dataType* to *DATE*
   - *caption* and *columnName* to `my_date`
2. Run the application. In the running Frame 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 Save changes to save the changes back to the text file.
3. View the text file in a text editor, such as Notepad. 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 = 8859_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.

Next, change the date pattern, edit the data, and save the changes again.

1. Close the application and the text files and return to the JBuilder 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 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, such as Notepad. 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 = 8859_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 grid in a different pattern, set the *displayMask* property. To modify data in the grid 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 from a TableDataSet to a text file" on page 8-5. 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 will be ignored on export.

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 6-14.

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 6-14.

Chapter

9

Using data modules to simplify data access

Data modules simplify data access development in your applications. Data modules offer you a centralized design-time 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.

A data module is a specialized container for data-access components. Once you define your *DataSet* components and their corresponding *Column* components in a data module, all frames that use the module have consistent access to the data sets and columns without requiring you to recreate them on every frame 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.

The *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 extend it by adding your data components.

When you create a data module and add any component that would automatically appear under the Data Access section of the Component tree (*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 box, select “DataModule1.database1” (or something similar) from the Database choice box.

Creating a data module using the designer tools

To create a data module,

1. Select File | Close to close any existing project (unless you want to add the data module to an existing project. In that case, skip this step).
2. Create a new project by selecting File | New Project. The Project Wizard displays.
3. Select the Browse button. You can create a data module in a separate project and once defined, refer to the data module class in your application. You can also create a data module in the same project as your application.
4. Choose a location and a name for the project. Choose Save.
5. Select File | New and double-click the Data Module icon.
6. 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.
7. Click the OK button to close the dialog box. The data module class is created and added to the project.

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.

Adding data components to the data module

To customize your data module using the UI Designer,

- Select the data module file in the Navigation pane.
- Select the Design tab of the Content pane to activate the UI Designer.

- 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 to add the *Database* component to the *DataModule*.
  3. In the Inspector, set the *connection* property (via the database *connectionDescriptor*).

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, "Accessing data."

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 and the next, your data module file will look like this:

```
package untitled7;

import java.awt.*;
import java.awt.event.*;
import borland.jbcl.layout.*;
import borland.jbcl.control.*;
import com.borland.dx.dataset.*;
import com.borland.dx.sql.dataset.*;
import java.lang.*;

public class DataModule1 implements DataModule{
  private static DataModule1 myDM;
  Database database1 = new Database();
  QueryDataSet queryDataSet1 = new QueryDataSet();
  public DataModule1() {
    try {
      jbInit();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
  private void jbInit() throws Exception{
    database1.setConnection(new  com.borland.dx.sql.dataset.ConnectionDescriptor
       ("jdbc:odbc:dataset tutorial",
       "SYSDBA", "masterkey", false, "sun.jdbc.odbc.JdbcOdbcDriver"));
    queryDataSet1.setQuery(new com.borland.dx.sql.dataset.QueryDescriptor(database1,
       "select FIRST_NAME, PHONE_EXT from employee", null, true, Load.ALL));
    queryDataSet1.addEditListener(new com.borland.dx.dataset.EditAdapter() {

         public void deleting(DataSet dataSet) throws Exception{
          queryDataSet1_deleting(dataSet);
        }
    );
  }
```

```
  public static DataModule1 getDataModule() {
    if (myDM == null)
      myDM = new DataModule1();
    return myDM;
  }
  public com.borland.dx.sql.dataset.QueryDataSet getQueryDataSet1() {
    return queryDataSet1;
  }
  void queryDataSet1_deleting(DataSet dataSet) throws Exception{
    //for example, business logic goes here
  }
}
```

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 others do not have these rights. 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,

- Select the data module file in the Navigation pane.
- Select the Design tab of the Content pane to activate the UI Designer.
- Select the component in the Component tree to which you want to add business logic.
- Select the Events page 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 and the data module.
2. Compile the data module class by selecting Run | Run. This creates the data module class files in the directory specified in Project | Properties, Output Root Directory.
3. Select File | Close.

To reference the data module in your application,

1. Select File | New Application. Accept all defaults. (Optionally, open an existing project using File | Open.)
2. Select the application's Frame file.
3. Import the package that the data module class belongs to (if it is outside your package) by selecting Wizards | Use Data Module.
4. Click the Browse For Class button to open the Package Browser. 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. Select the data module class. If you do not see the data module class here, check to make sure the project compiled without errors.
5. Click OK.

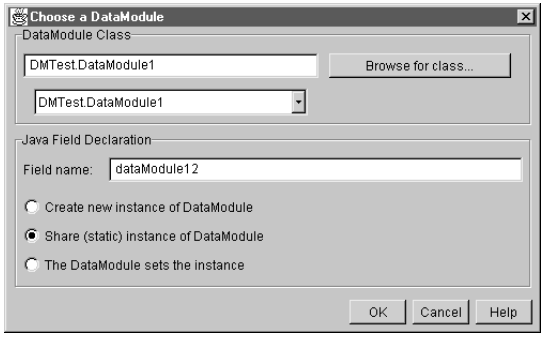
Click the Design tab to open the UI Designer; the instance of the data module appears in the Component Tree. Clicking the entry for the data module does not display its *DataSet* components nor its *Column* components. This is intentional to avoid modification of the business logic contained in the data module from outside.

When designing your application, you'll notice that the *dataSet* property of a UI control 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 Component Tree.

If you have a complex data model 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 box

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



- You can select a data module in two ways:
  - Select a data module from the drop-down list of data modules currently associated with the current project.
  - Click the Browse for Class button to open the Package Browser. A tree of all known packages and classes is displayed. If you do not see your *DataModule* class in this list, use 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 here in the Structure pane and in the Component tree. Select a name that describes the data in the data module.
- You can choose to Create New Instance Of DataModule, Share (Static) Instance of DataModule, or let The Data Module Set The Instance. 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 Share (Static) Instance of Data Module. If you only have a single *Frame* subclass in your application, select Create New Instance Of DataModule.
- 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 box 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:

```
employeeDataModule = untitled1.DataModule1.getDataModule();
```

If Create New Instance Of DataModule is selected, the following code will be added to the *jbInit()* method of the Frame file:

```
employeeDataModule = untitled1.DataModule1();
```

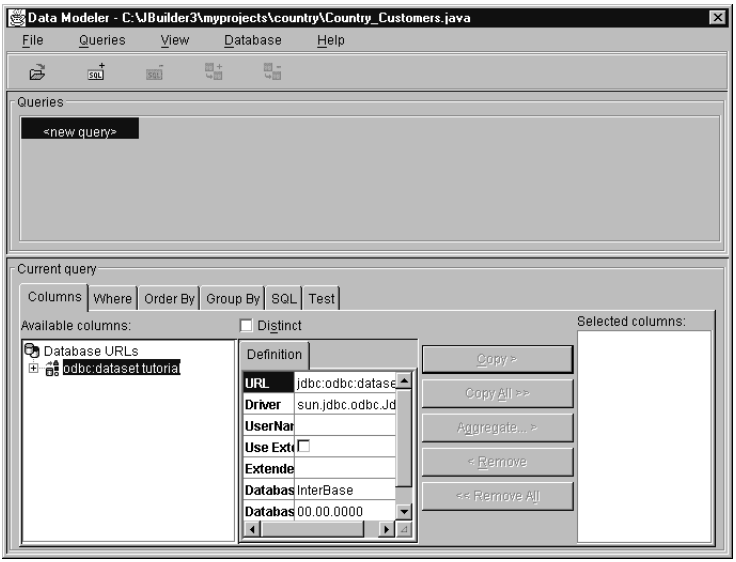
Using the Data Modeler to create a data module

The Data Modeler assists you in defining one or more SQL queries that connect to JDBC databases. It can also help you define master-detail relationships between queries. After you build your SQL queries, you can choose to store the result as a CORBA Interface Definition Language (IDL) file or as a JBuilder data module in a .java file.

To display the Data Modeler to begin a new query,

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

If you don't have a project open when you attempt to display the Data Modeler, the Project Wizard appears first. When you finish using the Project Wizard, the Data Modeler displays.



To open an existing IDL or Java data module in the Data Modeler,

1 Right-click the module in the Navigation pane.
2 Choose the Open with Data Modeler command.

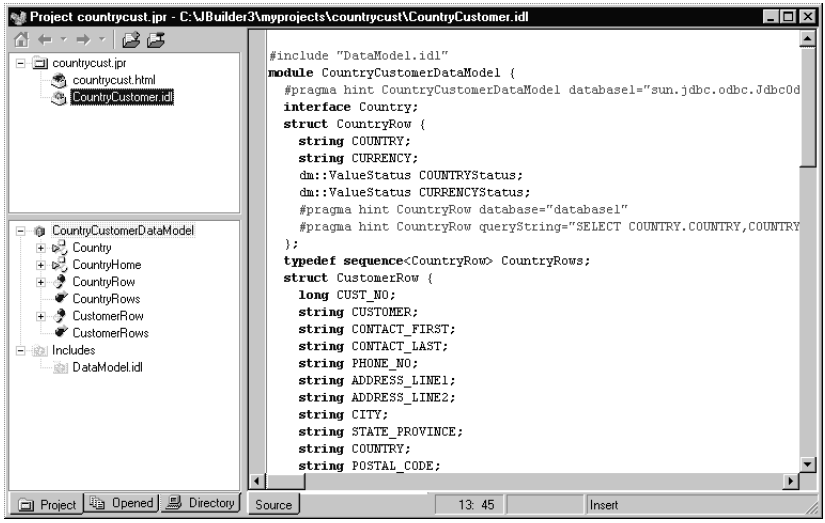
You can choose to save your query in an IDL file or as a JBuilder Java data module.

See the sections that follow "Creating an IDL file with the Data Modeler" on page 4-4 of *Developing distributed applications* for a discussion of building SQL queries using the Data Modeler.

To save the queries you built,

1 Choose File | Save in the Data Modeler and specify a name with an .idl extension for an IDL file or specify a name with a .java file for a Java data module.
2 Exit the Data Modeler.

The resulting file appears in your project. Select the file in the Navigation pane to view the code the Data Modeler generated.



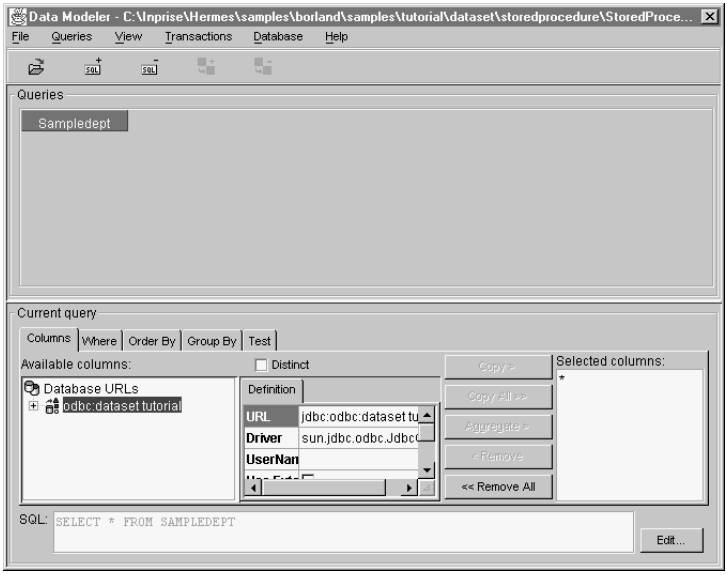
The lines that begin with #pragma hints are used by JBuilder tools. When you use the Data Modeler to redesign an .idl or .java data module, the Data Modeler uses the information contained in the #pragma hints. The Application Generator also uses them to generate database-driven applications.

Opening a data module in the Data Modeler

You can open an existing data module in the Data Modeler and add, delete, or edit the contents of the data module. To do this,

1. Open a project that contains a data module (for example, samples\com\borland\samples\dx\storedprocedure\storedprocedure.jpr).
2. Right-click on the data module file in the Navigation pane (for example, StoredProcedureDM.java).
3. Select Open With Data Modeler.

4 The Data Modeler displays, looking like this:



To learn how to add, edit, and delete information from the Data Modeler, see the Data Modeler online help, or see the sections that follow "Creating an IDL file with the Data Modeler" on page 4-4 of *Developing distributed applications* for a discussion of building SQL queries using the Data Modeler.

To save any changes you make,

1 Choose File | Save in the Data Modeler.

2 Exit the Data Modeler.

Chapter

10

Persisting and storing data in a DataStore

DataStore is a high-performance, small-footprint, 100% Pure 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 the *DataStore*s, refer to the *DataStore Programmer's Guide*.

When to use a DataStore

When to use a *DataStore*:

- **Organization**. To organize an application's *StorageDataSet*s, files, and serialized JavaBean/Object state into a single Pure 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. *StorageDataSet*s also provide excellent data binding support for data-aware UI controls.

- **Performance**. To increase performance and save memory for large *StorageDataSet*s. *StorageDataSet*s 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 *DataStore*s, refer to the *DataStore Programmer'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 8-5). 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 *DataStore Programmer's Guide*.

Chapter

11

Filtering, sorting, and locating data

Once you've completed the providing phase of your application and have the data in an appropriate DataExpress package *DataSet* component, you're ready to work on the core functionality of your application and its user interface. This chapter demonstrates the typical database application features of filtering, sorting, and locating data.

A design feature of the DataExpress package is that the manipulation of data is independent of how the data was obtained. Regardless of which type of *DataSet* component you use to obtain the data, you manipulate it and connect it to controls in exactly the same way. Most of the examples in this chapter use the *QueryDataSet* component, but you can replace this with the *TableDataSet* or any *StorageDataSet* subclass without having to change code in the main body of your application.

Each sample 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, 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 introductory tutorial, "An introductory database tutorial using a text file" on page 5-3.

All of the following examples and tutorials involve accessing SQL data stored in a remote database. These examples use the sample files included with Local InterBase Server. This data is accessed using JDBC and the JDBC-ODBC Bridge software. For instructions on how to setup and configure Local InterBase, JDBC and the JDBC-ODBC bridge, see Chapter 2, "Installing and setting up JBuilder for database applications."

If you're having difficulties running the database tutorials that connect to Local InterBase sample databases, the topic "Troubleshooting JDBC database connections in the tutorials" on page 2-8 provides common connection errors.

We encourage you to use the samples as guides when adding these functions to your application. Finished projects and Java source files are provided in the JBuilder samples directory (\samples\com\borland\samples\dx by default) for many of these tutorials, with comments in the source file where appropriate. All files referenced by these examples are found in the JBuilder samples directory, or in the InterBase examples directory.

To create a database application, you first need to connect to a database and provide data to a *Data*Set. "Providing data" on page 11-2 sets up a query that will be used for each of the following database tutorials. These options 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" on page 11-4. Filtering temporarily hides rows in a *DataSet*.
- "Sorting data" on page 11-7. Sorting changes the order of a filtered or unfiltered *DataSet*.
- "Locating data" on page 11-11. Locating positions the cursor within the filtered or unfiltered *DataSet*.

Providing data

This topic provides the steps for providing data to a *QueryDataSet* for use with the tutorials in this chapter. A query can be used to join tables, select only some rows, select only some columns, and sort. The query that will be used in these tutorials is:

```
SELECT EMP_NO, FIRST_NAME, LAST_NAME FROM EMPLOYEE WHERE EMP_NO < 10
```

This SQL statement selects only some columns `(EMP_NO, FIRST_NAME, LAST_NAME)` and some rows `(WHERE EMP_NO < 10)` from a table `(EMPLOYEE)`.

To query a database table,

1. Select File | Close from the menu. Select File | New from the menu. Double-click the Application icon and accept all defaults, except uncheck Use Only JDK & Swing Classes, to create a new application. Optionally, name the files to make referring to them easier in the following tutorials. In Step 2 of the Application Wizard, check Generate Status Bar. Click Finish.
2. Select the Frame file in the Navigation pane. Select the Design tab to activate the UI Designer.
3. Click the *Database* component on the Data Express tab of the Component palette, then click in the Component tree to add the component to the application.

   Open the *connection* property editor for the *Database* component by double-clicking the *connection* property in the Inspector. Set the connection properties to the Local InterBase sample employee table as follows:

Property name	Value
Connection URL	jdbc:odbc:DataSet Tutorial
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 in the gray area below the button. When the connection is successful, click OK.

4 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.

Double-click the *query* property of the *QueryDataSet* component in the Inspector and set the following properties:

Property name	Value
Database	*database1*
SQL Statement	`SELECT EMP_NO, FIRST_NAME, LAST_NAME FROM EMPLOYEE WHERE EMP_NO < 10`
Place SQL text in resource bundle	unchecked

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

To view the data in your application we need to add some UI components and bind them to the data set. To do this,

1 Add a *GridControl* component from the JBCL tab to the Designer. Set its *dataSet* property to *queryDataSet1*. You'll notice that the grid fills with data.

2 Add a *NavigatorControl* component from the JBCL tab to the Designer. Set its *dataSet* property to *queryDataSet1*. This will enable you to move quickly through the data set when the application is running, as well as provide a default mechanism for saving changes back to your data source.

3 A *StatusBar* control was added to the application in the Application Wizard. It displays at the bottom of the frame. Among other information, the status bar displays the current record. To bind it to the data set, select it in the Component tree, and set its *dataSet* property to *queryDataSet1*.

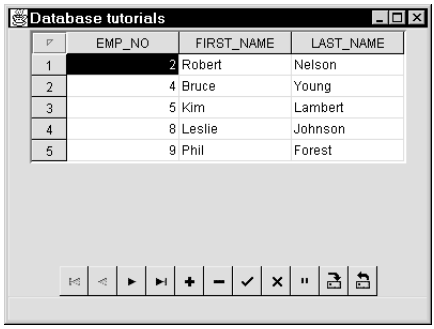
4 Select Run | Run to run the application and browse the data set.

The EMPLOYEE data set contains 42 records and 11 fields. In the status bar for this application, you will notice that the grid only contains 5 records and 3 fields. Remember that the SQL statement we used to create this data set limited the number of rows and columns we retrieved into the data set. To retrieve all records we could have used the SQL statement

```
SELECT * FROM EMPLOYEE
```

For more information on providing data to your application, see Chapter 5, "Accessing data." For information on creating this example using dbSwing components, see Chapter 13, "Using other controls and events."

The running application should look like this:



Filtering data

In JBuilder, data is extracted from a server into a data set. 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 account information by customer or in looking up a phone number by last name. 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 controls 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 12-2. You can change the order of a view by setting the *DataSet*'s *sort* property. For more information on sorting data, see "Sorting data" on page 11-7.

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 *TextFieldControl* that lets the user specify the column to filter. Then we create another *TextFieldControl* 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 *ButtonControl* 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\com\borland\samples\dx\FilterRows directory of your JBuilder installation under the project name FilterRows.jpr.

To create this application:

1. Create a new application, add a *Database* component, a *QueryDataSet* component (select * from employee), and a *GridControl* control (connected to the *queryDataSet1*) by following "Providing data" on page 11-2. This step enables you to connect to a database, read data from a table, and view and edit that data in a data-aware control.
2. Add two *TextFieldControl* components and a *ButtonControl* component from the JBCL tab. The *TextFieldControl* components enable you to enter a field and a value to filter on. The *ButtonControl* component executes the filtering mechanism.
3. Define the name of the column to be filtered and its formatter. To do this, select the Frame file in the Structure pane, then select the Source tab. Add this **import** statement to the existing **import** statements:

   ```
   import com.borland.jbcl.model.*;
   import com.borland.dx.dataset.Variant;
   ```

4. 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;
   ```

5. 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*'s *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* or *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, select the Frame file in the Structure pane, then select the Design tab. Select the *queryDataSet1*, select the Events tab of the Inspector, select the *filterRow* event, and double-click the value box. A *RowFilterListener* is automatically generated at the end of your file. It calls a new method in your class, called *queryDataSet1_filterRow* method. Add the filtering code to this event. You can copy the code from the

online help by selecting the code and clicking the Copy button (*Ctrl+C* will not copy the contents of the Help Viewer).

```
void queryDataSet1_filterRow(ReadRow readRow, RowFilterResponse rowFilterResponse) {
   try {
      if (formatter == null || columnName == null || columnValue == null ||
         columnName.length() == 0 || columnValue.length() == 0)
         // user set field(s) are blank, so add all rows
         rowFilterResponse.add();
      else {
         readRow.getVariant(columnName, v);  // fetches row's value of column
         String s = formatter.format(v);     // formats this to a string
                                              // true means show this row
         if (columnValue.equals(s))
            rowFilterResponse.add();
         else rowFilterResponse.ignore();
      }
   }
   catch (Exception e) {
      System.err.println("Filter example failed");
   }
}
```

6 Override the *actionPerformed* event for the *ButtonControl* to retrigger the actual filtering of data. To do this, select the Frame file in the Navigation pane, select the Design tab, select the *ButtonControl*, and click the Events tab on the Inspector. Select the *actionPerformed* event and double-click the value box for its event.

The Source tab displays the stub for the *buttonControl1_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.

```
void buttonControl1_actionPerformed(ActionEvent e) {

   try {

   // Get new values for variables that the filter uses.
   // Then force the data set to be refiltered.

      columnName = textFieldControl1.getText();
      columnValue = textFieldControl2.getText();
      Column column = queryDataSet1.getColumn(columnName);
      formatter = column.getFormatter();

      // Trigger a recalc of the filters

      queryDataSet1.refilter();

      // The grid should now repaint only those rows matching these criteria
      }
      catch (Exception ex) {
         System.err.println("Filter example failed");
      }
   }
}
```

7 Compile and run the application.

To filter data, define the column you wish to filter (Last_Name as initially defined) in the first *TextFieldControl*, define the value you wish to filter for in the second *TextFieldControl* (Young as initially defined), and then press the *ButtonControl*. Leaving either the column name or the value blank removes any filtering and allows all values to be viewed.

Example: Filtering with a restrictive clause in a query

In "Tutorial: Adding and removing filters" on page 11-5, data was extracted from a server to a data set and additional filtering was applied to the data in the data set. You can also use a filter to restrict the data being fetched from the server into the data set. To do this, you can use a WHERE clause in your SQL statement. In this example, we view all employees that reside in Japan,

1. Select Frame in the Structure pane and then select the Design tab of the Content pane.
2. Drop a *Database component*, a *QueryDataSet* component, and a *GridControl* component on the Component tree.
3. Set the *connection* property for the *Database* component as follows (assuming your system is set up as in Chapter 2, "Installing and setting up JBuilder for database applications"):
   - Connection URL: jdbc:odbc:DataSet Tutorial
   - Username: SYSDBA
   - Password: masterkey
4. Select the *query* property for the data set. In the *query* custom property editor, set Database to *database1*. Enter the following text in the SQL Statement text box:

   ```
   SELECT * FROM employee WHERE JOB_COUNTRY = "Japan"
   ```

5. Run the application by selecting Run | Run.

This example could be modified to code the query with a parameter variable instead of a constant. For more information on using parameterized queries, see "Using parameterized queries to obtain data from your database" on page 5-23.

Sorting data

Sorting a data set defines an index that allows the data to be displayed in a sorted order without actually reordering the rows in the table on the server.

Data sets can be sorted on one or more columns. When a sort is defined on more than one column, the data set is sorted

- 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 and have the exact same value
- 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 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 "Sorting and indexing" on page 11-9 for further discussion of indexes.

Sorting data in a GridControl

If your application includes a *GridControl* or a *JdbTable* that is associated with a *DataSet*, you can sort on a single column in the grid by clicking the column header in the running application. Click again to toggle from ascending to descending order. When using the *GridControl*, you can disable this feature by setting the *sortOnHeaderClick* property of the *GridControl* to false.

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*'s *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-useable 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:

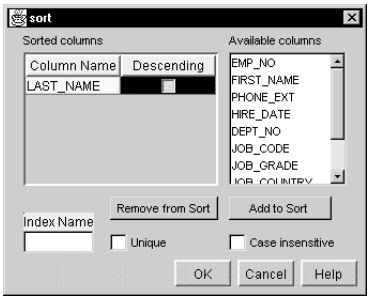
1. Open or create the project you created for "Providing data" on page 11-2.
2. Select the *QueryDataSet* in the Component tree. Click the Design tab.

3 In the Inspector, 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.

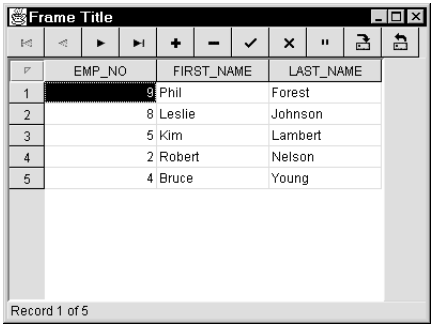
6 Uncheck the Descending option to view names in ascending (A..Z) order. The dialog will look like this:



7 Click the OK button.

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

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



Sorting and indexing

There are two options on the Sort dialog that benefit from further discussion: Unique and Indexes. Sorting and indexing are closely related. The following describes unique 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, any duplicate rows are moved to another 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 created 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 maintains the sort orders (that is, indexes), and the unique constraint is enforced, 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 must be to let you revert to a previously defined sort. The data set 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 an existing named index, specify the index name in the *sortDescriptor*.

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:

```
queryDataSet1.setSort(null);
```

Locating data

A basic need of data applications is to find specified data. With the JBCL, you can perform two types of locates:

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

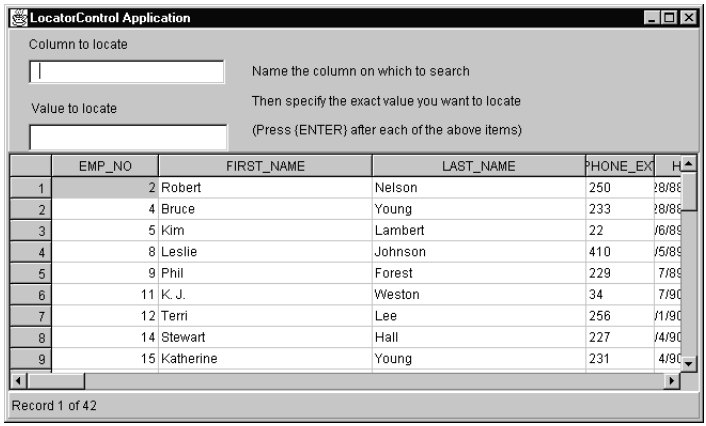
Locating data with the LocatorControl

The JBCL includes a *LocatorControl* component that provides locate functionality in a user-interface control. The *LocatorControl* includes an incremental search feature for *String* type columns. Its *columnName* property specifies which column to perform the locate in. If not set, the locate is performed on the last column visited in a UI control (for example the *GridControl*).

If you include the JBCL *StatusBar* component in your application, *LocatorControl* prompts and messages are displayed on the status bar. Connect both the *StatusBar* and the *LocatorControl* to the same *DataSet* component to enable this feature.

The samples\com\borland\samples\dx\Locator directory of JBuilder includes a finished example of an application that uses the *LocatorControl* under the project name Locator.jpr. This sample shows how to set a particular column for the locate operation as well as using a *TextFieldControl* component to prompt the user for the

column to locate in. The completed application (with the application window reduced in size) looks like this:



To create this application,

1. Create a new project and application files using the Project and Application Wizards. Select the Generate status bar and Center frame on screen options from Step 2 of the Application Wizard.
2. Open the Designer by highlighting the *Frame* file and selecting the Design tab at the bottom of the AppBrowser.
3. Add the following components to the *Frame*. They are available from the Data Express tab of the Component Palette.
   - *Database*
   - *QueryDataSet*
4. Add the following components to the *Frame* class. They are available from the JBCL tab of the Component Palette.

   This topic shows how to create a UI for your application using JBCL components. To create a UI using dbSwing components, see Chapter 13, "Using other controls and events."

   - *GridControl*
   - *LocatorControl*
   - *TextFieldControl* (add this component if you want to allow the user to specify which column to locate in)
   - Two *LabelControl* components. Both provide on-screen instructions for the user, one for the *TextFieldControl*, the other for the *LocatorControl*.

   Check the screen shot of the running application (shown above) to see the approximate positioning of each component.

Set the properties of the components as follows (also described in more detail in as described in"Providing data" on page 11-2):

1. Set the *connection* property of the *Database* component to the InterBase sample files.
2. Set the *query* property of the *QueryDataSet* component as follows:

Property name	Value
Database	database1
Query String	select * from employee

3. Set the *dataSet* property of the *LocatorControl*, *GridControl*, and *StatusBar* to queryDataSet1.
4. Select the *TextFieldControl*. Select the Events tab of the Inspector. Select the *TextFieldControl* component's *keyPressed()* event and add the following code to allow the user to specify which column to locate in.

```
if (e.getKeyCode() == KeyEvent.VK_ENTER) {
    locatorControl1.setColumnName(textField1.getText());
    locatorControl1.requestFocus();
 }
```

   This code tests for the *Enter* key being pressed. If it determines that the *Enter* was pressed, the *columnName* property for the *LocatorControl* is set to the column named in the *TextFieldControl*. This instructs the *LocatorControl* to perform locates in the specified *Column*. Focus is then shifted to the *LocatorControl* so that you can enter the value to search for.

5. Set the *text* property of the *LabelControl* that you placed beside the *TextFieldControl* if you want to prompt the user for the column to locate on:

```
Column to locate
```

   Alternatively, if you want to locate only in a particular *Column*, set the *LocatorControl* component's *columnName* property to the *DataSet* column you want to locate on, for example, LAST_NAME.

6. Set the *text* property of the *LabelControl* that you placed beside the *LocatorControl* to:

```
Value to locate
```

Note See the screen shot of the running application earlier in this section for additional instructional text.

7. Run the application.

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

- Enter the column name that you want to perform the locate on in the *TextFieldControl*. Press *Enter*.

Note In this sample, there is no validation of the column name, so be sure to enter a correctly spelled column name.

- Start typing the value to locate in the *LocatorControl*. If you're locating in a *String* column, as you type, notice that the *LocatorControl* 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.

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 11-15.)
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 occurences, call *locate()* again, specifying a different locate option, for example, *Locate.NEXT* or *Locate.LAST*. See *Locate* class variables in the *DataExpress Library Reference* 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 (instantiatable) 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 11-15.) If a match is found, the current row position moves to that row. All data-aware controls 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 using *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 in the JBCL *model* package (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 earlier in this topic.

Chapter

12

Adding functionality to database applications

Once you've completed the providing phase of your application and have the data in an appropriate DataExpress package *DataSet* component, you're ready to work on the core functionality of your application and its user interface. Chapter 11, "Filtering, sorting, and locating data", introduced sorting, filtering, and locating data in a data set. This chapter demonstrates other typical database applications.

A design feature of the DataExpress package is that the manipulation of data is independent of how the data was obtained. Regardless of which type of *DataSet* component you use to obtain the data, you manipulate it and connect it to controls in exactly the same way. Most of the examples in this chapter use the *QueryDataSet* component, but you can replace this with the *TableDataSet* or any *StorageDataSet* subclass without having to change code in the main body of your application.

Each sample is created using the JBuilder 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, 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 "An introductory database tutorial using a text file" on page 5-3 in this manual.

All of the following examples and tutorials involve accessing SQL data stored in a remote database. These examples use the sample files included with Local InterBase Server. This data is accessed using JDBC and the JDBC-ODBC Bridge software. For instructions on how to setup and configure Local InterBase, JDBC and the JDBC-ODBC bridge, see Chapter 2, "Installing and setting up JBuilder for database applications."

If you're having difficulties running the database tutorials that connect to Local InterBase sample databases, "Troubleshooting JDBC database connections in the tutorials" on page 2-8 provides common connection errors.

We encourage you to use the samples as guides when adding these functions to your application. Finished projects and Java source files are provided in the JBuilder samples directory (\samples\com\borland\samples\dx by default) for many of these tutorials, with comments in the source file where appropriate. All files referenced by these examples are found in the JBuilder samples directory, or in the InterBase examples directory.

To create a database application, you first need to connect to a database and provide data to a *DataSet*. "Providing data" on page 11-2 sets up a query that can be used for each of the database tutorials in this chapter.

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* component, and set its *storageDataSet* property to the *StorageDataSet* object that contains the data you want to view,

1. Open or create the project you created for"Providing data" on page 11-2.
2. Select the Frame file in the Navigation pane. Click the Design tab.
3. Add a *DataSetView* component from the Data Express tab to the Component tree.
4. In the Inspector, set the *storageDataSet* property of the *DataSetView* component to *queryDataSet1*.
5. The *DataSetView* navigates independently of its associated *StorageDataSet*. Add another *GridControl*, *StatusBar*, and *NavigatorControl*, or their dbSwing equivalents, to the UI Designer. To enable the controls to navigate together, set their *dataSet* properties to *dataSetView1*. To create this application using dbSwing components, see "Creating a database application UI using dbSwing components" on page 13-2.
6. 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 Navigation 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 11-7 for more information on the *sortDescriptor*.
   - Select the *masterLink* property to define a parent data set for this view. See "Defining a master-detail relationship" on page 7-2 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 11-4 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*.

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 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 in the samples\com\borland\samples\intl directory of your JBuilder installation.

To override the default formats for numeric and date values that are stored in locale files, set the *displayMask*, *editMask*, or *exportDisplayMask* property (as appropriate) on the *Column* component of the *DataSet*.

The formatting capabilities provided by DataExpress package string-based patterns are typically sufficient for most formatting needs. If you have more specific formatting needs, the format mechanism includes general-purpose interfaces and classes that you can extend to create custom format classes.

Display masks

Display masks are string-based patterns that are used to format the data displayed in the *Column*, for example, in a *GridControl*. 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 *StatusBar* control when the *StatusBar* and the UI control that displays the data for editing (for example, a *GridControl*) 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 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.

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 in the samples\com\borland\samples\intl directory of your JBuilder installation.

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 preceeded 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 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 Library Reference.*

For example, the code that sets the display and edit masks to the first pattern listed below is:

```
column1.setDisplayMask(new String("00000{-9999}"));
column1.setEditMask(new String("00000{-9999}"));
```

The following table explains some pattern specifications:

Pattern specification	Data values	Formatted value	Meaning
00000{-9999}	950677394 00043 1540001	95067-7394 00043 00154-0001	Display leading zeros for the left 5 digits (required), optional remaining characters include a dash literal and 4 digits. Use this pattern for U.S. postal codes.
L0L 0L0	H2A2R9 M1M3W4	H2A 2R9 M1M 3W4	The *L* specifies any letter A–Z, entry required. The *0* (zero) specifies any digit 0–9, entry required, plus (+) and minus (–) signs not permitted. Use this pattern for Canadian postal codes.
{(999)} 000-0000^!;0	4084311000	(408) 431-1000	A pattern for a phone number with optional area code enclosed in parenthesis. The carat (^) positions the cursor at the right side of the field and data shifts to the left as it is entered. To ensure data is stored correctly from right to left, use the ! symbol. (Numeric values do this automatically.) The zero (0) indicates that literals are not stored with the data.

Patterns for boolean data

The *BooleanFormat* component uses a string-based pattern that is helpful when working with values that can have two values, stored as **true** or **false**. Data that falls into each category is formatted using string values you specify. This formatter also has the capability to format **null** or unassigned values.

For example, you can store gender information in a column of type **boolean** but can have JBuilder format the field to display and accept input values of "Male" and "Female" as shown in the following code:

```
column1.setEditMask("Male;Female;");
column1.displayMask("Male;Female;");
```

The following table illustrates valid boolean patterns and their formatting effects:

Pattern specification	Format for true values	Format for false values	Format for null values
male;female	male	female	(empty string)
T,F,T	T	F	T
Yes,No,Don't know	Yes	No	Don't know
smoker;;	smoker	(empty string)	(empty string)
smoker;nonsmoker;	smoker	nonsmoker	(empty string)

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 12-10. For more information on working with columns, see "Working with columns" on page 5-41.

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 12-11.

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 12-16.

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

Tutorial: Creating a calculated column in the designer

This tutorial builds on the example in "Querying a database" on page 5-13. 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\com\borland\samples\dx\CalcColumn directory of your JBuilder installation under the project name CalcColumn.jpr.

1. Complete the example, or open the sample project QueryProvide.jpr.
2. Select Frame1.java from the Navigation pane. Select the Design tab of the UI Designer. Double-click *queryDataSet1* in the Component tree to open the Columns Editor. Click the "+" sign in the Columns Editor toolbar, and set the following properties in the Inspector for the new column:

Property name	Value
calcType	*calculated*
caption	NEW_SALARY
Property name	Value
calcType	*calculated*

   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 grid in the designer. The calculations are only visible when the application is running. The data type of BIGDECIMAL is used here because that is the data type of the SALARY column which will be used in the calculation expression. Calculated columns are always read-only.
3. Select the *queryDataSet1* object, go to the Events tab of the Property Inspector, select the *calcFields* event handler, and double-click its value. This creates the stub for the event's method in the Source window.
4. Modify the event method to calculate the salary increase, as follows:

```
void queryDataSet1_calcFields(ReadRow readRow, DataRow dataRow, boolean boolean1)
   throws DataSetException{
   //calculate the new salary
   dataRow.setBigDecimal("NEW_SALARY",
      readRow.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

(*readRow*, or *changedRow*), an output row for putting any changes you want to make to the row (*dataRow*, or *calcRow*), and a boolean (*boolean1*, or *isPosted*) that indicates whether the row is posted in the *DataSet* or not. You may not want to recalculate fields on rows that are not posted yet.

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

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

6 Run the application to view the resulting calculation expression.

When the application is running, the values in the calculated column will automatically adjust to changes in any columns referenced in the calculated expression.

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* property 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*, or a custom aggregation class that you define.

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 *TextFieldControl* 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\com\borland\samples\dx\AggCalc directory of your JBuilder installation under the project name AggCalc.jpr.

1. Select File | Close from the menu to close existing applications. Select File | New. Double-click the Application icon to start the Application Wizard. Accept all defaults to create a new application.
2. Select Frame1.java in the Navigation pane, then select the Design tab of the UI Designer to put JBuilder into Design mode.
3. Put a *Database* component from the Data Express tab of the Component palette on the Component Tree and set its *connection* property as described in "Providing data" on page 11-2.

   Click the Test Connection button to test the connection and ensure its validity. If the connection is successful, click OK. If not successful, see "Troubleshooting JDBC database connections in the tutorials" on page 2-8.
4. Add a *QueryDataSet* component from the Data Express tab to the Component Tree. This will form the query to populate the data set with values to be aggregated. Set the *query* property of *queryDataSet1* as follows:

For this option	Make this choice
Database	*database1*
SQL Statement	`select cust_no, PO_NUMBER, SHIP_DATE, TOTAL_VALUE from SALES`
Place SQL text in resource bundle	unchecked

   Click the Test Query button to test the query and ensure its validity. If successful, click OK. If not successful, review the SQL statement for errors.
5. Add a *GridControl* component from the JBCL tab of the Component palette and set its *dataSet* property to *queryDataSet1*. This enables us to view data in the designer and when the application is running.

   This topic shows how to create a UI for your application using JBCL components. To create a UI using dbSwing components, see "Creating a database application UI using dbSwing components" on page 13-2.
6. Double-click *queryDataSet1* in the Component tree to open the Columns Editor. Click the "+" sign in the Columns Editor toolbar, and set the following properties in the Inspector for the new column:

Property name	Value
caption	`GROUP_TOTAL`
columnName	`GROUP_TOTAL`
currency	`True`
dataType	*BIGDECIMAL*
calcType	*aggregated*

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 more code, use *Alt+Z* to toggle between the standard view and an expanded view of code.

```
column1.setCurrency(true);
column1.setCalcType(com.borland.dx.dataset.CalcType.AGGREGATE);
column1.setCaption("GROUP_TOTAL");
column1.setColumnName("GROUP_TOTAL");
column1.setDataType(com.borland.jbcl.util.Variant.BIGDECIMAL);
```

7 Add a *TextFieldControl* from the JBCL tab of the Component palette to the UI Designer. Set its *columnName* property to GROUP_TOTAL. Set its *dataSet* property to *queryDataSet1*. This control displays the aggregated data. You may wish to add a *LabelControl* to describe what the text field is displaying.

No data will be displayed in the *TextFieldControl* 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.

8 Select <*new column*> again and set the following properties in the Inspector for the new column:

Property name	Value
columnName	PO_NUMBER
dataType	*STRING*

This and the next two steps ensure the columns that will display in the grid are persistent. Persistent columns are enclosed in brackets in the Component tree. Also, when you add more than one column, you can manually edit the *setColumns()* method in the Source code pane to change the order of display of the columns.

9 Select <*new column*> again and set the following properties in the Inspector for the new column:

Property name	Value
columnName	CUST_NO
dataType	*INT*

10 Select <*new column*> again and set the following properties in the Inspector for the new column:

Property name	Value
columnName	SHIP_DATE
dataType	*TIMESTAMP*

11 Select the GROUP_TOTAL column in the Component tree. To define the aggregation for this column, double-click on the *agg* property to display the *agg* property editor.

In the *agg* property editor,

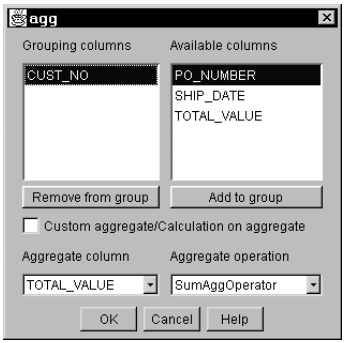
- Double-click CUST_NO in the Available Columns list to select this as the field that will be used to define the group.
- Select TOTAL_VALUE from the Aggregate Column list to select this as the column that contains the data to be aggregated.
- Select *SumAggOperator* from the Aggregate Operation list 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.

This step generate 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()));
```

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



12 Run the application by selecting Run | Run to view the aggregation results.

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 *TextFieldControl* will display the aggregation for the CUST_NO of the currently selected row.

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 uses CMT to extract 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; there should be no way to edit the contents of either list. Note that 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*, obtained also from CMT. Although the current set of *AggOperator*s 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 *AggOperator*s built into DataExpress package as well as any user-defined *AggOperator*s detectable by CMT 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.

Clicking the OK button assigns an *AggDescriptor* created with the settings defined in the dialog box to the *Column*'s *agg* property. Clicking the Cancel button restores the Column's agg property to its previous state. Clicking the Help button activates the Help system, displaying "Agg property editor" as the current help page.

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 totalling columns in a line items table (like SALES). The dollar amounts can be totalled 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 *DataSet*s. You may wish to create aggregation classes for calculating an average, standard deviation, or variance.

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 as a result of a lookup or via a picklist.

- "Tutorial: Creating a lookup using a calculated column" on page 12-17

  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 specify the criteria in the *pickList* property editor. 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. A tutorial for creating this type of lookup field follows.

  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 11-11.

  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: Looking up choices with a picklist" on page 12-19

  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 further in this topic 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 control, such as a grid, when the application is running.

See also "Working with columns" on page 5-41

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\com\borland\samples\dx\Lookup directory of your JBuilder installation.

For more information on using the *calcFields* event to define a calculated column, refer to "Using calculated columns" on page 12-9.

This application is primarily created in the UI Designer.

1. Select File | Close from the menu. Select File | New from the menu. Double-click the Application icon. Accept all defaults.
2. Select the Frame file from the Navigation pane, then select the Design tab to begin adding components.
3. Add a *Database* component by clicking on the Database component from the Data Express tab and then clicking in the Component tree. In the Inspector, set its *connection* property as follows:

For this option	Make this choice
Connection URL	`jdbc:odbc:DataSet Tutorial`
Username	`SYSDBA`
Password	`masterkey`

Test the connection to ensure its validity. If not successful, see "Troubleshooting JDBC database connections in the tutorials" on page 2-8.

4 Add a *QueryDataSet* component from the Data Express tab of the palette to the Component tree. 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 *queryDataSet1* as follows:

For this option	Make this choice
Database	*database1*
SQL Statement	`select * from EMPLOYEE_PROJECT`
Place SQL text in resource bundle	unchecked

Test the query to ensure its validity. When successful, click OK.

5 Add another *QueryDataSet* component. This forms the query that provides the looked up data to the lookup field. Set the *query* property of *queryDataSet2* as follows:

For this option	Make this choice
Database	*database1*
SQL Statement	`select EMP_NO, FIRST_NAME, LAST_NAME from EMPLOYEE`
Place SQL text in resource bundle	unchecked

Test the query to ensure its validity. When successful, click OK.

6 Add a *GridControl* and set its *dataSet* property to *queryDataSet1*. This will enable you to view data in the designer and when the application is running.

This topic shows how to create a UI for your application using JBCL components. To create a UI using dbSwing components, see "Creating a database application UI using dbSwing components" on page 13-2.

7 In the Component tree, click the + sign to the left of the *queryDataSet1* component to expose all of the columns. Select <*new column*> and set the following properties in the Inspector for the new column:

Property name	Value
calcType	*lookup*
caption	`EMPLOYEE_NAME`
columnName	`EMPLOYEE_NAME`
dataType	*STRING*

The new column will display in the list of columns and in the grid 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 grid 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.

8 Select the *pickList* property in the Inspector. The pickList property editor opens. Set the following properties to look up EMP_NO in *QueryDataSet2* and display the appropriate LAST_NAME field in the new column.

Field	Value
Picklist/Lookup Dataset	*queryDataSet2*
queryDataSet2	EMP_NO
queryDataSet1	EMP_NO
Lookup column to display	LAST_NAME

When you click OK, you will see the correct last name displayed in the grid.

9 Select Run | Run to run the application.

When the application is running, the values in the calculated lookup column will automatically adjust to changes in any columns, in this case the EMP_NO column, referenced in the calculated values. If the EMP_NO field is changed, the lookup will display the value associated with the current value when that value is posted.

Tutorial: Looking up choices 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\com\borland\samples\dx\Picklist directory of your JBuilder installation.

This application is primarily created in the designer.

1 Select File | Close from the menu. Select File | New from the menu. Double-click the Application icon to create a new application. Accept all defaults.

2 Select the Frame file from the Navigation pane, then select the Design tab to begin adding components.

3 Add a *Database* component from the Data Express tab of the palette to the Component tree. Set its *connection* property as follows:

For this option	Make this choice
Connection URL	`jdbc:odbc:DataSet Tutorial`
Username	`SYSDBA`
Password	`masterkey`

Test the connection to ensure its validity. If not successful, see "Troubleshooting JDBC database connections in the tutorials" on page 2-8.

4 Add a *QueryDataSet* component from the Data Express tab of the palette to the Component tree. This will form the query to populate the list of choices. Set the *query* property of *queryDataSet1* as follows:

For this option	Make this choice
Database	*database1*
SQL Statement	`select COUNTRY from COUNTRY`
Place SQL text in resource bundle	unchecked

Test the query to ensure its validity. When successful, click OK.

5 Add another *QueryDataSet* component. This will form the query to populate the grid with information from the EMPLOYEE table. Set the *query* property of *queryDataSet2* as follows:

For this option	Make this choice
Database	*database1*
SQL Statement	`select EMP_NO, FIRST_NAME, JOB_COUNTRY from EMPLOYEE`
Place SQL text in resource bundle	unchecked

Test the query to ensure its validity. When successful, click OK.

6 Add a *GridControl* from the JBCL tab of the palette to the UI Designer. Set its *dataSet* property to *queryDataSet2*.

7 Add a *NavigatorControl* from the JBCL tab of the palette to the UI Designer. Set its *dataSet* property to *queryDataSet2*.

8 Click the + sign to the left of the *queryDataSet2* component in the Component tree to expose all of the columns. Select *JOB_COUNTRY*.

9 Double-click the *pickList* property in the Inspector to bring up the *pickListDescriptor*. Set the *pickList* properties as follows:

Property name	Value
Picklist/Lookup Dataset	*queryDataSet1*
queryDataSet1	COUNTRY
Data Type	STRING
Display Column?	checked
queryDataSet2	JOB_COUNTRY

Click OK.

10 Run the application by selecting Run | Run.

When the application is running, you can select a row in the grid and pick a country from the 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>.

Specifying required data in your application

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 control are determined at runtime 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 runtime 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 runtime error when access to the column's data fails.

Making columns persistent

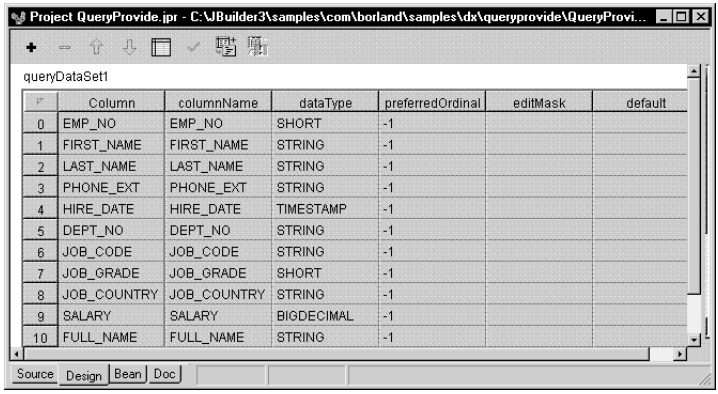
You can make a column persistent by setting any property at the *Column* level (for example, an edit mask). When a column has become persistent, square brackets ([ ]) are placed around the column name.

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\com\borland\samples\dx directory of your JBuilder installation. (This directory is available only if you selected the Typical option from the JBuilder setup program, or selected Sample files from the Custom option.)

2 Select the *Frame* file and click the Design tab from the bottom of the right pane of the AppBrowser.

3 Double-click the *DataSet* object from the Component tree. This displays the Column Designer for the data set. The Column Designer looks like this for the Local InterBase employee sample table:



	Column	columnName	dataType	preferredOrdinal	editMask	default
0	EMP_NO	EMP_NO	SHORT	-1		
1	FIRST_NAME	FIRST_NAME	STRING	-1		
2	LAST_NAME	LAST_NAME	STRING	-1		
3	PHONE_EXT	PHONE_EXT	STRING	-1		
4	HIRE_DATE	HIRE_DATE	TIMESTAMP	-1		
5	DEPT_NO	DEPT_NO	STRING	-1		
6	JOB_CODE	JOB_CODE	STRING	-1		
7	JOB_GRADE	JOB_GRADE	SHORT	-1		
8	JOB_COUNTRY	JOB_COUNTRY	STRING	-1		
9	SALARY	SALARY	BIGDECIMAL	-1		
10	FULL_NAME	FULL_NAME	STRING	-1		

4 Select the *Column* that you want to set the property of. 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 other 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 grid 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 grid.
- 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 controls, such as a grid control.
- The Choose Properties button lets you choose which properties to display in the Designer.
- The Invoke 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 the *DataStore Programmer's Guide*.

  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 runtime. See "Using the Column Designer to persist metadata" on page 5-43.
- The Make All MetaData Dynamic button will REMOVE CODE from the source file. See "Making metadata dynamic using the Column Designer" on page 5-44.

You could also do all of your column manipulation through the Column Designer. You can right-click on the header of the grid to select (or unselect) a column property. That property is added to, or removed from, the grid. This in convenient for setting global properties for every column in a data set, for example, *width*.

To close the Column Designer, double-click on any UI component in the Component tree, or single-click on a different component, and select Activate Designer. In other words, 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 "Working with columns" on page 5-41.

Storing Java objects

DataSet and *DataStore* can store Java objects in columns of a *DataSet*.

Fields in a SQL table, reported by JDBC as being of type *java.sql.Types.OTHER*, are mapped into columns whose data type is*Variant.OBJECT*, or you can set a column's data type to Object and set/get values through the normal data set API.

If a *DataStore* is used, the objects must be serializable. If they are not, an exception is raised whenever the *DataStore* attempts to save the object. Also, the class must exist on the CLASSPATH when it attempts to read an object. If not, the attempt will fail.

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.

Chapter

13

Using other controls and events

To create a database application, you first need to connect to a database and provide data to a *DataSet*. “Providing data” on page 11-2 sets up a query that can be used as a starting point for any of the discussions in this chapter.

JBuilder provides many data-aware versions of JFC components. Many of these components are located on the JBCL tab of the Component palette. To use these 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.
5. 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 JBCL components that can be made aware of data in a data set:

- CheckBoxControl
- ChoiceControl
- FieldControl
- LabelControl
- ListControl
- NavigatorControl
- GridControl
- ImageControl
- StatusBar
- TextAreaControl
- TextFieldControl

The following list provides a few of the dbSwing components available from the dbSwing tab of the Component palette. for information on how to use these

components in your database application, see “Creating a database application UI using dbSwing components” on page 13-2.

- JdbTable
- JdbNavToolBar
- JdbList
- JdbTextPane
- JdbTextField
- TableScrollPane

When should I use JBCL components and when should I use dbSwing (JFC) components?

JBuilder 3.0 offers you a choice of dbSwing, Swing, and JBCL components. 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.

If you programmed with JBCL in the past, you can certainly continue to do so. It is possible to mix JBCL and dbSwing components, as they both respond to the same *DataSet* events. But you should probably avoid doing so as you might run into unanticipated problems. Using only dbSwing components, you can be sure all your components are lightweight. For a discussion of lightweight versus heavyweight components, see “Lightweight components” in Part III, *Creating JavaBeans*.

JBCL components provide backward compatibility for existing applications and are more functional in certain applications, for example, those that use edit or display masks.

Creating a database application UI using dbSwing components

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. Some noteworthy dbSwing components are:

- *JdbComboBox* for displaying multi-column pick list data.
- *JdbNav* components for navigating *DataSet*s.
- *DBPasswordPrompter* for prompting for *Database* password and user names.
- *DBEventMonitor* for tracing *DataSet* events.
- *DBDisposeMonitor* for releasing *DataSet*resources and automatically closing *DataStore*s.
- *DBExceptionHandler* for closing *DataStore*s when exceptions occur.

- Data-binder and model components for making almost any Swing-derivable component data-aware.

Tutorial: Using dbSwing components to create a database application UI

Most of the tutorials in this book use JBCL components to create a database application UI. For any of the tutorials that use the JBCL components, such *GridControl*, *StatusBar*, and *NavigatorControl*, you can substitute this section and use dbSwing components to create the UI for your application. The dbSwing package reference contains a table comparing JBCL and dbSwing controls.

To use dbSwing components to view and navigate data in your application,

1. Set up a database application, for example, an application that connects to a database and runs a query, for example, "Providing data" on page 11-2.
2. Add a *JdbNavToolBar* component from the dbSwing tab at the top of the frame in the Designer. Unlike its JBCL counterpart, *JdbNavToolBar* automatically attaches itself to *DataSet*s (whichever *DataSet* has focus), so you do not need to set its *dataSet* property.

   This will enable you to move quickly through the data set when the application is running, as well as provide a default mechanism for saving changes back to your data source.
3. Add a *JdbStatusLabel* control from the dbSwing tab at the bottom of the frame in the Designer. Unlike its JBCL counterpart, *JdbStatusLabel* automatically attaches itself to *DataSet*s (whichever *DataSet* has focus), so you do not need to set its *dataSet* property.

   Among other information, the status bar displays information about the current record or current operation.
4. Add a *TableScrollPane* component from the Swing Containers tab to the Designer.

   Scrolling behavior is not available by default in any Swing component or dbSwing extension, so, to get scrolling behavior, we add the Swing or dbSwing component to a *TableScrollPane*.
5. Drop a *JdbTable* component from the dbSwing tab into the *TableScrollPane* component. Set its *dataSet* property to *queryDataSet1*.

   You'll notice that the Designer displays a grid that fills with data. *JdbTable* allows multi-line column headers. Enter a column caption like "line1\nline2" to see both lines in the table's header. Also, you can sort data by clicking on a table header in the running application.
6. Select Project | Properties. On the Run/Debug page, check Execution Log. Click OK.

   Using the Execution Log instead of the Console Window enables you to view a list of activities as they occur, while running or debugging. You can also type notes into this window, and save the contents of the window to a text file. To open this window, choose View | Execution Log.
7. Select Run | Run to run the application and browse the data set.

To save changes back to the data source, you can use the Save button on the navigator tool or, for more control on how changes will be saved, see Chapter 6, "Saving changes back to your data source" for information on using other data resolving mechanisms.

There are some sample applications that use dbSwing components. The samples are located in the samples\com\borland\samples\dbswing directory of your JBuilder installation

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. The JBCL includes a *StatusBar* control 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 *StatusBar*. This string overwrites the existing contents of the *StatusBar* and is overwritten itself when the next string is written to the *StatusBar*.

You can also connect the *StatusBar* control to a *DataSet*. The *StatusBar* control 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 StatusBar control

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

You can add a *StatusBar* control to your application in two ways:

- Have the Application Wizard add the *StatusBar* automatically when creating the structure for your application.

  To include a *StatusBar* when creating your application:

  1. Select File | Close.
  2. Select File | New. Double-click the Application icon.
  3. Enter the appropriate values in the Project Wizard. Click Finish.
  4. Enter appropriate values in the first page of the Application Wizard.
  5. Click the Next button. The second page appears.

6 Place a checkmark beside the Generate Status Bar option. Enter appropriate values for the remaining options on this page.

7 Click Finish to generate the application files. The *Frame* object will automatically include the *StatusBar* control.

- Add the *StatusBar* control manually to the UI of your application *Frame*.

  To add the *StatusBar* to the UI of your existing application:

  1 Open the project files for the application you want to add a *StatusBar* to. If you do not have an application, use the files created from "Providing data" on page 11-2.

  2 Select the *Frame* file in the Navigation pane of the AppBrowser, then click the Design tab that appears at the bottom of the AppBrowser.

  3 Click the JBCL tab of the Component Palette and select the *StatusBar* component button:

  4 Draw the *StatusBar* below the *GridControl* component.

Note The exact location of the *StatusBar* may change depending on the layout manager in use. For more information on layouts, see "Using layout managers" in the online manual, *Building Applications with JBuilder*.

  5 Set the *dataSet* property of the *StatusBar* control to the *DataSet* object whose status messages you want displayed in the *StatusBar*. You typically connect a *StatusBar* control to the same *DataSet* as the UI control (such as a grid) which displays the data from the *DataSet*. This sets both controls to track the same *DataSet* together and is often referred to as a *shared cursor*.

  Once you set the *dataSet* property, you'll notice that the *StatusBar* control displays information that the cursor is on Row 1 of x (where x is the number of records in the *DataSet*).

  6 Double-click the *QueryDataSet* in the Component Tree. 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.

  7 Run the application.

Running the StatusBar 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 *StatusBar* 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 that doesn't meet the minimum value. When you attempt to move off the row, the *StatusBar* displays the same message that the row cannot be posted due to invalid or missing field values.

By setting the text of the *StatusBar* at relevant points in your program programmatically, you can overwrite the current message displayed in the *StatusBar* 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 grid, validation errors when editing data, and so on.

For other examples of applications that include a *StatusBar* control, see the following topics:

- *TextDataFile* tutorial in "An introductory database tutorial using a text file" on page 5-3.
- Real-world database application development in the "Sample international database application" on page 17-2.

Synchronizing visual controls

Several data-aware controls can be associated with the same *DataSet*. In such cases, the controls navigate together. When you change the row position of a control, the row position changes for all controls that share the same cursor. This synchronization of controls 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. Controls 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 controls by setting each of their *dataSet* properties to the same data set. When controls 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 *NavigatorControl* and a *GridControl* in your program, and connect both to the same *QueryDataSet*, clicking the "Last" button of the *NavigatorControl* automatically displays the last record of the *QueryDataSet* in the *GridControl* as well. If those controls are set to different *dataSet* components, they do not reposition automatically to the same row of data.

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 control

If you set the *dataSet* property on a control, you should avoid accessing the *DataSet* data or model information programatically through the control until the control's peer has been created; basically, this means until the control is displayed in the application UI.

Operations which fail or return incorrect/inconsistent results when executed before the control is displayed in the application UI include any operation that accesses the model of the control's architecture. This may include,

- *<control>.get()* or *<control>.set()* operations
- *<control>.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 control and its model architecture are properly initialized.

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 JBCL and dbSwing data-aware controls. 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 JBCL and dbSwing data-aware controls, 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 *StatusBar* 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 *StatusBar* 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 bar.

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 JBCL and dbSwing controls and *DataSet* objects, create your own handler class and connect it to the *ExceptionEvent* listener of the *DataSetException* class.

Most of the events in the *dataset* package throw a *DataSetException*. This is very useful when your event handlers use *dataSet* APIs (which usually throw *DataSetException*). This releases you from coding **try**/**catch** logic for each event handler you write. Currently the JBuilder design tools do not insert the "throws DataSetException" clause in the source java code it generates, however you can add the clause yourself.

Chapter

14

Creating a distributed database application

This chapter discusses creating a distributed database application using the *DataSetData* component and Remote Method Invocation (RMI) for creating a distributed application.

Chapter 15, "Creating database applications with the Data Modeler and Application Generator" discusses using the CORBA method for creating a distributed application. Chapter 7, "Exploring Java RMI-based distributed applications in JBuilder" in *Developing distributed applications* discusses creating a non-database distributed application using Java RMI.

Creating a distributed database application using DataSetData

In the samples\com\borland\samples\dx directory of your JBuilder Enterprise edition installation is a sample project, DataSetData.jpr, that 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 Local InterBase 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 grid. Editing performed on the client can be saved using a *NavigatorControl*'s Save button.

For more information on developing distributed database applications, see Chapter 1, "Developing distributed applications" in *Developing distributed applications*. In particular, Chapter 4, "Building distributed applications with JBuilder" in *Developing distributed applications* contains an outline for creating a distributed database application using the Data Modeler and Application Generator,

and Chapter 7, "Exploring Java RMI-based distributed applications in JBuilder" in *Developing distributed applications* contains a tutorial for developing a Java RMI-based distributed application in JBuilder. Some distributed applications features and documentation are only available in certain versions of JBuilder.

For more information on writing custom providers, see "Writing a custom data provider" on page 5-38. For information on writing or customizing a resolver, see "Customizing the default resolver logic" on page 6-14.

Understanding the sample distributed database application (using Java RMI and DataSetData)

The sample project found in samples\com\borland\samples\dx\DataSetData.jpr 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 grid 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. Install a version of JBuilder containing this sample.
2. Open this application in JBuilder by selecting File | Open and browsing to samples\com\borland\samples\dx\datasetdata\datasetdata.jpr.
3. Start the RMI registry by selecting Tools | RMI Registry from JBuilder. When the registry is running, an icon for its DOS window appears on the Taskbar. To close the RMI Registry, select it from the Taskbar, and enter *Ctrl+C*.
4. Select the file *DataServerApp* in the Navigation pane. Select Run | Run.
5. Select the file *ClientApp* in the Navigation pane. Select Run | Run.

These steps enable the *DataServerApp* to register itself as a service for RMI. *DataServerApp* will respond to two inquiries: *provideEmployeeData* and *resolveEmployeeChanges*, as defined in the file *EmployeeApi.java*. Both of these methods are implemented in the file *DataServerApp.java*.

The *ClientApp* file is a frame with a *GridControl*, a *NavigatorControl*, and a *TableDataSet*. Data is provided to the *DataSet* via a custom *Provider*, and data is saved to the source via a custom *Resolver*. The *Provider* gets the data from the *DataServerApp*. The *Resolver* uses the *DataServerApp* to resolve the changes back to the database. This is a multi-tier solution.

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.

Modifying the application to a 3-tier application

To modify the application to a 3-tier application,

- Select *DataServerApp.java* in the Navigation pane. Modify the database connection to a remote connection you have access to. Recompile.
- Run the RMI Registry (by selecting Tools | RMI Registry) and the *DataServerApp* from a remote machine you are connected to.

- Select *ClientFrame.java* in the Navigation pane. Select the Design tab. In the Component tree, select *clientProvider1* and modify the *hostName* property to the hostname shown on the *DataServerApp* running on the remote machine.
- Select *clientResolver1* in the Component tree. Modify the *hostName* property to the hostname shown on the *DataServerApp* running on the remote machine.
- Run the RMI Registry and the *ClientApp* on this machine.

For more information

- Read the RMI Documentation on the JavaSoft Web site at http://www.javasoft.com/products/jdk/1.1/docs/guide/rmi/index.html.
- Learn more about writing custom *Providers* and *Resolvers* by viewing the sample data set application samples\com\borland\samples\dx\providers.jpr. (Client/Server version only)
- Learn more about creating distributed applications in Chapter 1, "Developing distributed applications" in *Developing distributed applications*.
- Learn more about creating RMI -based applications in Chapter 7, "Exploring Java RMI-based distributed applications in JBuilder" in *Developing distributed applications*.

Chapter

15

Creating database applications with the Data Modeler and Application Generator

JBuilder provides you with the tools that can help you quickly create applications that query a database. The Data Modeler can build the queries for you and store it in a Java data module. If you store the queries in a Java data module, the Application Generator can then use the data module to generate a complete application for you.

Creating the 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,

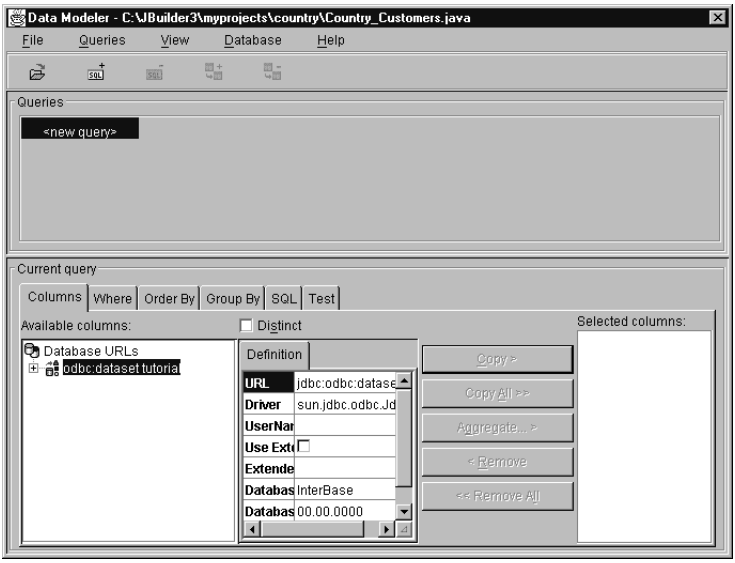
1. Choose New | Project to start the Project Wizard.
2. Fill in the fields in the dialog box.
3. Click the Finish button.

For more specific information about creating projects, see "Creating and Managing Projects" in *Building Applications with JBuilder* in the online Help.

To display the Data Modeler,

1. Choose File | New.
2. Double-click the Data Module icon and check the Invoke Data Modeler option if it isn't already selected.

If you have a project open, the Data Modeler appears. If you don't have a project open, the Project Wizard appears first. When you finish using the Project Wizard, the Data Modeler appears.



To open an existing Java data module in the Data Modeler,

1 Right-click it in the navigation pane.

2 Choose the Open With Data Modeler command.

To begin building an SQL query, double-click the URL that accesses your data or single-click the plus sign (+). Or select the URL and choose the Data Modeler's Database | Open Connection URL.

Adding a URL

If the database you want to access is not available from the Data Modeler, add the appropriate URL:

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.

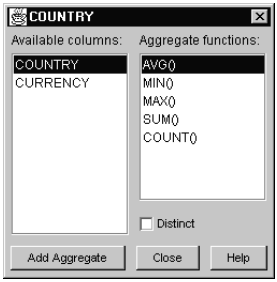
3 Type in the URL or use the Browse button to select the URL of the data you want to access.

Beginning a query

Begin building a query by selecting columns you want to add to the query, or by selecting an aggregate function that operates on a specific column:

- To add one or more columns,
  1. Click a column you want to view 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.
- To add an aggregate function to the query,
  1. Click the Aggregate > button to display a dialog box.



  2. Click the column whose data values you want aggregated in the Available Columns list.
  3. Click the function you want to use on that column from the Aggregate Functions column.
  4. If you want the function to operate on only unique values of the selected column, check the Distinct check box.
  5. Choose Add Aggregate to add the function to your query.

As you select columns and add functions, your SQL SELECT statement is being built. To view it, click the SQL tab.

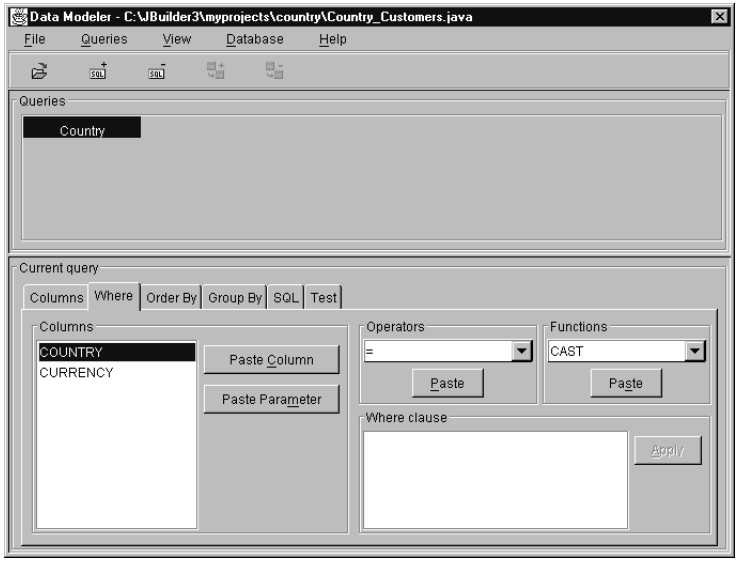
Selecting rows with unique column values

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

To add a WHERE clause to your SQL query, click the Where tab to display the Where page.



The Columns list on the left contains the columns of tables in the currently selected query in the Queries panel of the Data Modeler. Use the Columns, Operators, and Functions lists to build the 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'
```

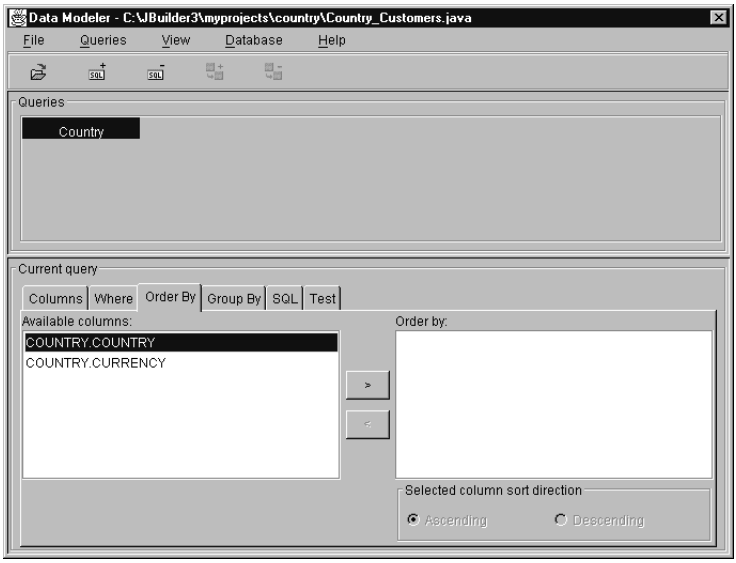
You would select and paste the COUNTRY column and the = operator. To complete the query, you would type in the data value directly, which in this case is ‘USA’.

When you are satisfied with your WHERE clause, click the Apply button. The WHERE clause is added to the entire SQL query. To view it, click the SQL tab.

Adding an Order By clause

To specify how rows of a query are sorted,

1. Select the query you want sorted in the Queries panel.
2. Click the Order By tab in the Current Query panel.



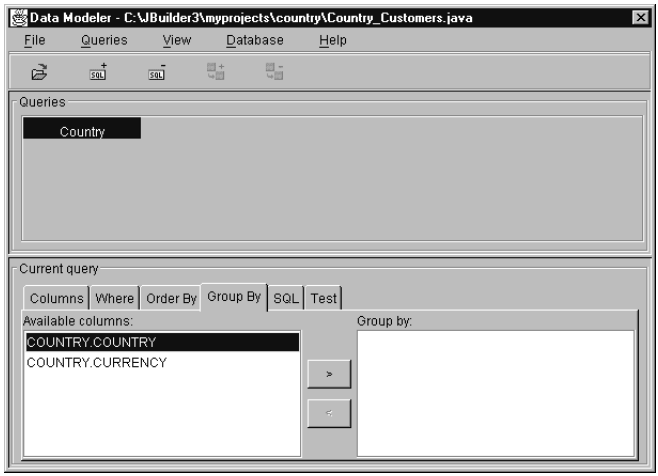
3. Select the column you want the query sorted by in the Available Columns box and click the button with the > symbol on it to transfer that column to the Order By box.
4. Select the sort order direction from the Selected Sort Order Direction options.

   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.

Adding a Group By clause

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 query. To view it, click the SQL tab.

Viewing and editing the query

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.

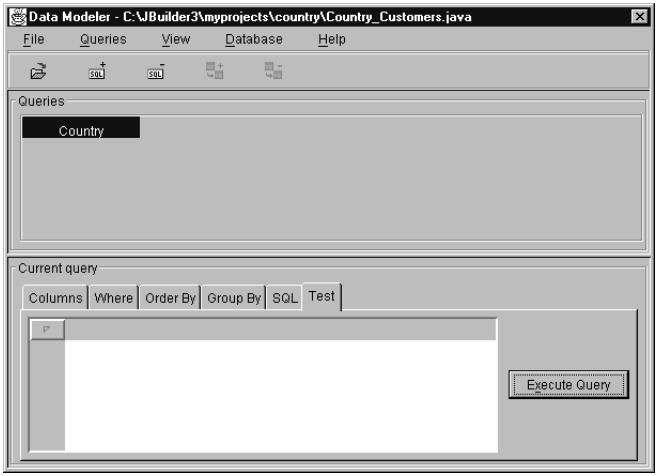
Testing your query

You can view the results of your query in the Data Modeler.

To see the results of the query you are building,

1 Click the Test tab.

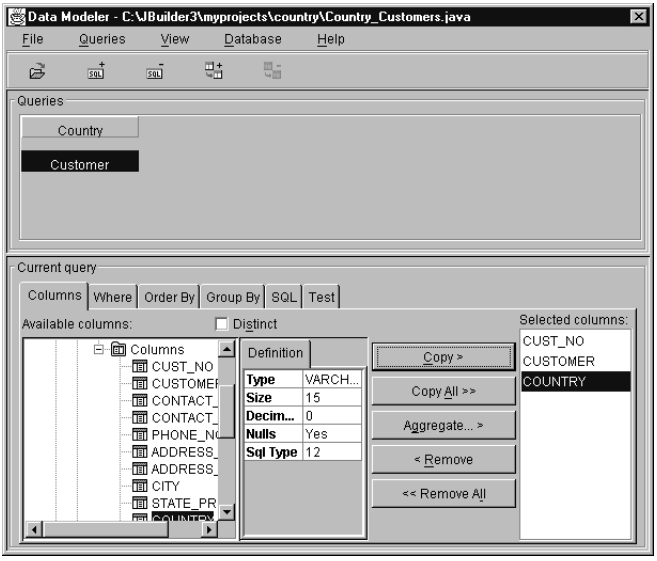
2 Click the Execute Query button.



If your query is a parameterized query, a Specify Parameter Values 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.

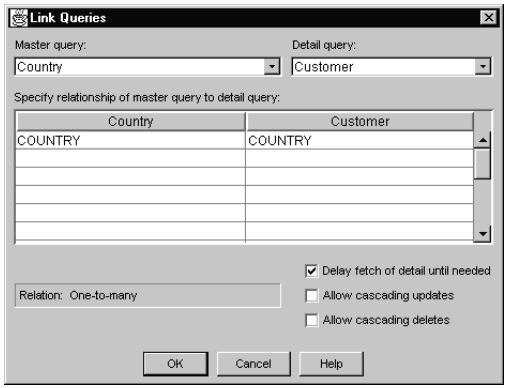


Specifying a master-detail relationship

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, 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 grid to visually specify the columns that link the master and detail queries together:

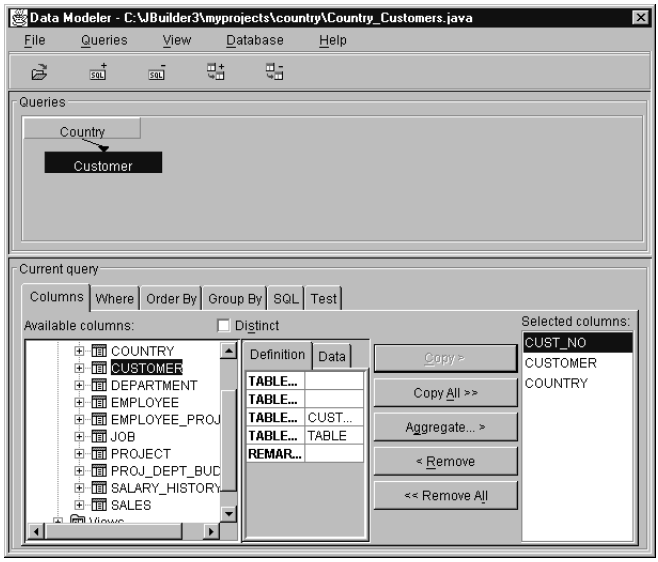
1 Click the first row under the master query column of the grid to display a drop-down list of all the specified columns in the master table. Select the column you want the detail data to be grouped under.

2 Click the first row under the detail query column of the grid 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.

You can use the grid to link multiple rows in the master table with rows in the detail table.

When the Link Queries dialog box closes, an arrow is shown between the two queries in the Queries panel showing the relationship between them.

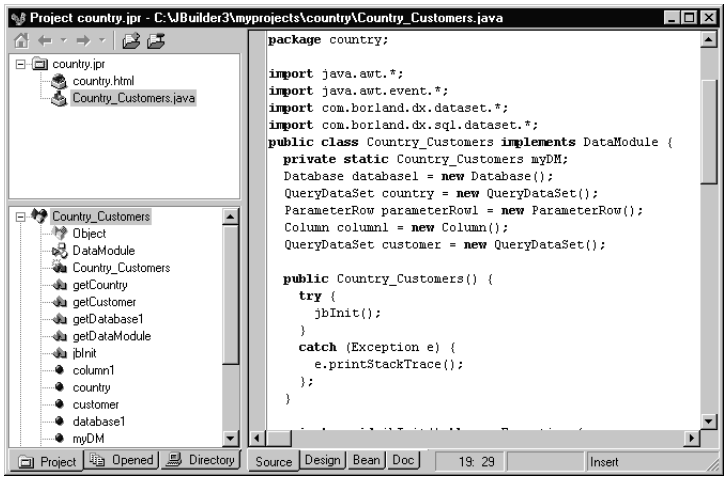


Saving your queries

To save the queries you built,

1. Choose File | Save in the Data Modeler and specify a name with a .java extension to save a JBuilder data module.
2. Exit the Data Modeler.

The resulting file appears in your project, and is selected in the navigation pane.

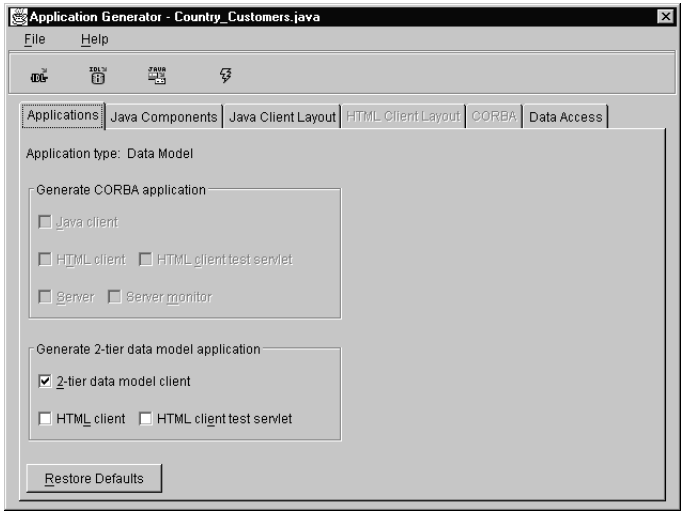


Generating database applications with the Application Generator

From your data module, JBuilder can generate two-tier client-server applications with its Application Generator. The Application Generator can generate applications from any JBuilder data module, not just those created with the Data Modeler. The following sections apply to both types.

To display the Application Generator, select one of these methods:

- Choose Yes in the dialog box that appears when you exit the Data Modeler and are asked if you want to invoke the Application Generator.
- Select the Application From Data Module icon in the Object Gallery:
  1. Choose File | New and select the Applications tab.
  2. Double-click the Application From Data Module icon.
  3. Select the data module you want to generate an application from in the dialog box that appears. You can select any JBuilder data module that you have or you can select one that was created by the Data Modeler.
  4. Choose OK and the Application Generator appears.
- Use the navigation pane context menu:
  1. Right-click the data module in the navigation pane.
  2. Select the Generate Application menu item.



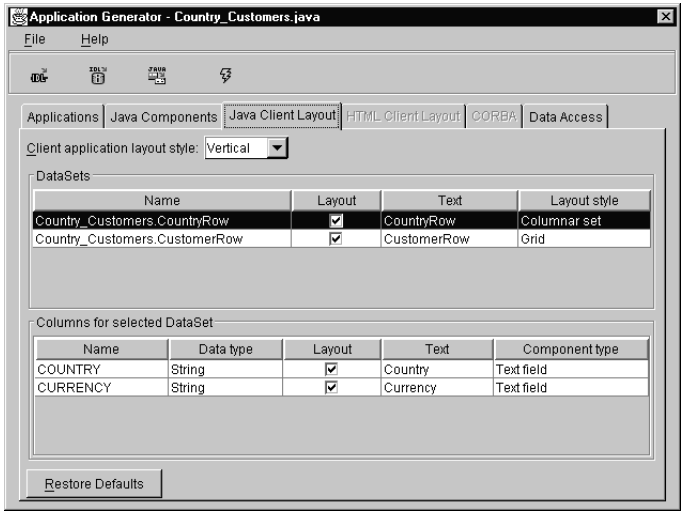
Preparing to generate the application

On the Applications page of the Application Generator, you select the type of application you want generated. For a two-tier database application, check the 2-tier Data Model Client option.

To prepare to generate an HTML client, check both the HTML Client and HTML Client Test Servlet options. When the test servlet option is selected, a test servlet and an HTML form are generated. When it is run, the test servlet verifies that all the libraries that are needed to run the servlet are correctly installed on the Web Server. For more information on configuring the Web Server, see "Deploying the client files (production environment)" on page 5-8 of *Developing distributed applications*.

Specifying a Java client layout

You can specify several of the parameters that determine the layout and appearance of your user interface. Click the Java Client Layout tab to display the Java Client Layout page.



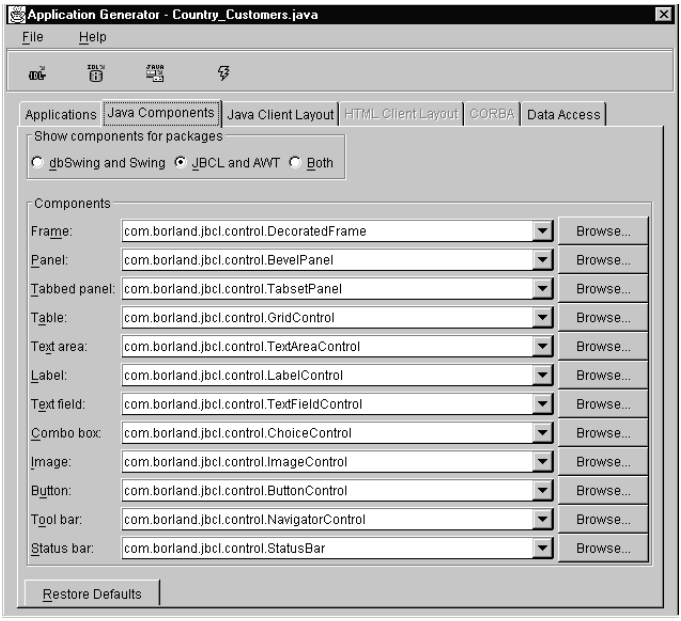
The Client Application Layout drop-down list determines how the datasets referenced in the data module are presented in the user interface of your client. You can select a horizontal, vertical, or tabbed layout. A horizontal layout displays generated components side by side, a vertical layout displays the components top to bottom, and a tabbed layout displays each component that displays a structure on a separate page of a tabbed panel.

The table in the center of the Java Client Layout page lists datasets available to your client application. Those that will appear in the client layout have the Layout check box checked. You can specify the layout style used to display them. Click a cell in the Layout Style column to display a drop-down list and select one of the layout styles. If you choose not to have the element appear in the client layout, uncheck the Layout check box.

The table at the bottom of the Java Client Layout page lists the columns of a dataset you selected in the Data Modeler. You can choose to hide a column by unchecking the Layout option and you can change the text used to identify the column in your client's user interface.

Specifying the controls used in the client user interface

You can exert further control over the appearance of your Java client UI using the Java Components page of the Application Generator. To display this page, click the Java Components tab.



The page displays a list of components your generated client UI might use. You can specify which component is used for the various parts of your UI.

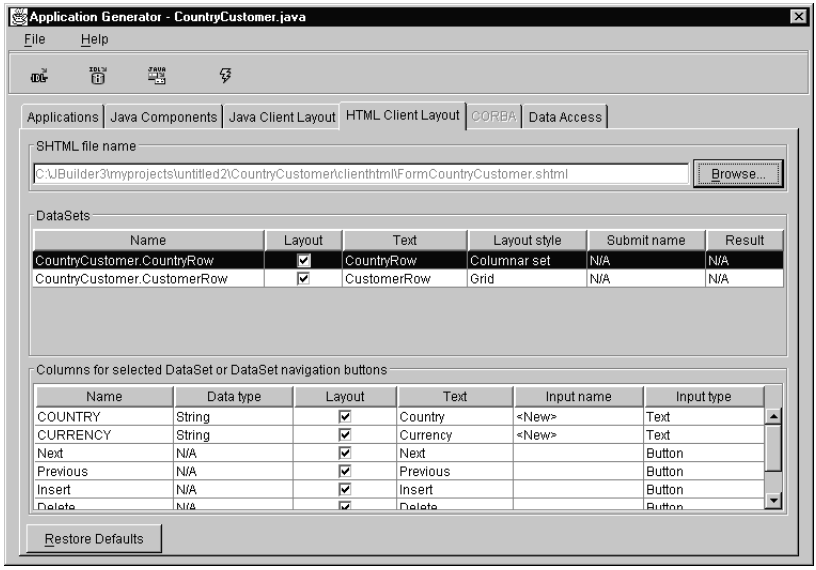
At the top of the page is the Show Components for Packages option. If you want to be able to select dbSwing and Swing components only for your UI elements, check the dbSwing and Swing option. If you prefer to use JBCL and AWT components, select that option. Or you can select Both to have dbSwing and Swing components as well as JBCL and AWT components to choose from.

In the Components box you specify which component to use to display each UI element. Open the drop-down lists and select your preferred component for each element. If you prefer to use a component other than the standard dbSwing-Swing or JBCL-AWT components, click the Browse button and use the Package browser that appears to locate the component you want to use.

Specifying an HTML client layout

To enable the HTML Client Layout page of the Application Generator, check the HTML Client box on the Applications page of the Application Generator. If an HTML file with the same name as the data module already exists, this box is checked and the page is enabled.

To display the HTML Client Layout page, click the HTML Client Layout tab.



The HTML Client Layout page enables you to generate HTML forms connected to columns in a dataset, or to load any HTML form and link the controls within it to dataset columns. The tags added to the HTML files are generic and the files are free of special tags to allow editing by any HTML tool. The mechanism that is used to link the controls to columns is the SERVLET tag. When the form is rendered, the servlet tag is placed at the end of each form. The object invoked by this servlet tag is the *gateway object*. The servlet tag generates JavaScript that appears on the client HTML page. If JavaScript is used, the HTML browser must be Netscape Navigator 4.0, Microsoft Internet Explorer 4.0, or a later version of these two browsers.

To specify the HTML client layout,

1. Specify an existing shtml file to be retrofitted with links to dataset columns, accept the default name for the shtml file, or modify the name and directory for the shtml file.
2. Select one of the DataSets in the first table. The columns for the selected dataset appear in the table at the bottom of the page.
3. Make the changes you want to the column layout in the table at the bottom of the page. You can uncheck the Layout check box if you don't want a column to appear. You can also change the text that labels the column by modifying the Text field. The Input Type specifies the type of control used to enter data for that column. If the SHTML file specified above already contains this information, it will display as specified and can be modified here.

For more complete information about creating HTML clients, see Chapter 5, "Creating, running, and deploying an HTML client" of *Developing distributed applications*.

Setting data access options

To set data access options, click the Data Access tab.

Treat Binary Array Data As Image Data option

When the Treat Binary Array Data As Image Data check box is checked, elements that are of binary array data are displayed as images; that is, the component becomes an Image on the Java Client Layout page and an Image component displays the data in your Java client. The Image component used is the one specified on the Java Components page. If the data module loaded into the Application Generator was created by the Data Modeler, this check box is on by default.

Make Generated Data Module Extend Source option

When the Generated Data Module Extend Source option is checked, the Application Generator generates a data module in the entities package and uses the original data module as a base class. If this option is not checked, it still generates a data module in the entities package but the data module contains no reference to the original data module.

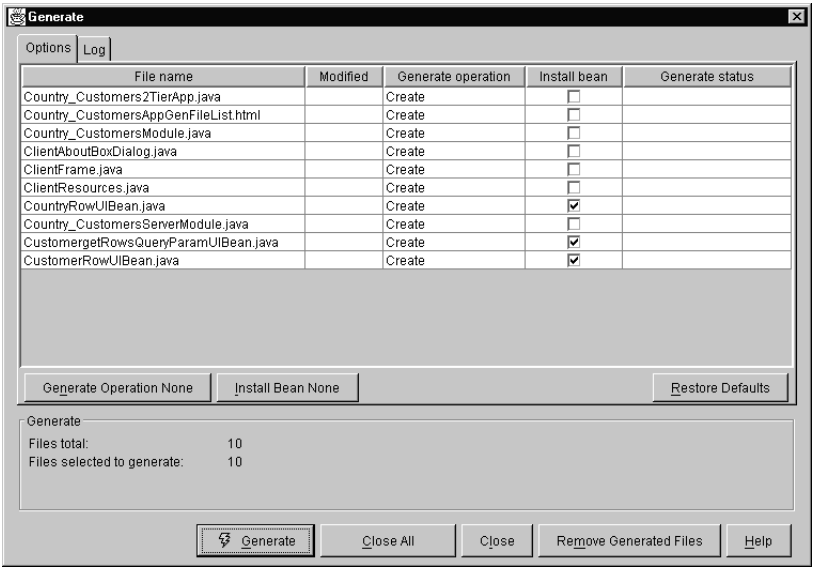
Changing the user name and password

You can change the user name and password in the table; select the current entries and change them to your new choices.

Generating the application

When you're ready to generate your distributed application, choose File | Generate or click the Generate icon on the Application Generator toolbar.

A Generate dialog box appears showing you the files the Application Generator will generate.



You can choose to exclude some files by selecting None as the Generate option from the drop-down list that appears when you highlight the current operation next to the file name. If you want to generate only a few files, click the Generate Operation None button, and the operation for all the files is specified as None. Then select Create from the drop-down list as the operation you want for the files to be generated. Once you click the Generate Operation None button, it becomes the Generate Operation All button. Clicking the Generate Operation All button changes the Generate Operation for all files to Create or Update.

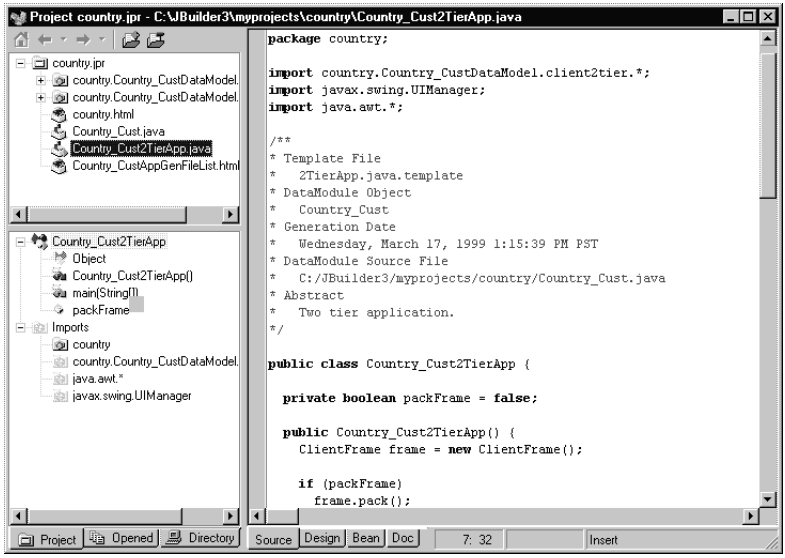
Once you modify any generated files, those files are checked as modified the next time you use the Application Generator. The Application Generator won't recreate files checked as modified by default, but you can force regeneration by changing the Generate Operation from None to Create.

Many of the files to be generated are JavaBeans. Those that have UIBean as part of their names are visual JavaBeans you might want to use to build other applications. You can choose to install them on a project page of the JBuilder component palette. If a component is set to be installed, the Install Bean check box for that file name is checked. If you choose to not install it, uncheck the check box.

If you choose to install none of the beans, click the Install Bean None button at the bottom of the page. Once clicked, the Install Bean None button becomes the Install Bean All button. Clicking this button checks all of the UI beans, so that they will be installed on the component palette.

To generate your application, click the Generate button. When the file generation completes, close the Application Generator.

Click the Log tab of the Generate dialog box to see the results. The newly created files appear in the navigation pane. Most of the generated files are grouped into two packages: *client2tier* and *entities*. If you specified an HTML client, there is also a *clienthtml* package. The Application Generator also generates a database application called *<ModuleName>DataModel2TierApp*.



The Application Generator also creates an HTML file that contains a list of all the files generated. Clicking a file name in the generated document displays that file in the Content pane.

For more information about the files the Application Generator generates, see "Examining the generated files" on page 4-23 of *Developing distributed applications*.

Once you've generated the files for your database application, you're ready to compile and run it for the first time. To compile your project, right-click the project node in the navigation pane and choose Make.

To run the application within JBuilder, choose Run | Run. Or if you can right-click a the generated application file and choose Run, which automatically compiles and then runs the application.

For information about using the Data Modeler and Application Generator to create multi-tier CORBA applications that access remote databases, see Chapter 4, "Building distributed applications with JBuilder" of *Developing distributed applications*.

Using a generated data module

Once you've created a data module with the Data Modeler and the Application Generator, you can use it in applications that you write. Follow these steps:

- In the source code of the frame for your application, add a *setModule()* method that identifies the data module with the Use Data Module wizard. 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.
- In the source code of your application file, call the frame's *setModule()* method, passing it an implementation of the data module.

For example, suppose you have used the Data Modeler and Application Generator to create a data module called *CountryDataModelModule*, which exists in the *entities* package. 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,

- Choose Wizards | Use Data Module while the frame's source code is visible in the editor.
- Specify the data module you want to use with the wizard.
- Select the Application Sets The Instance By Calling setModule() option.
- Choose OK.

The resulting code of the frame would look like this:

```
package untitled3;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import untitled3.CountryDataModel.entities.*;

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(&quot;Frame Title&quot;);
  }

//Overriden 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 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 class that implements the data module interface.

The code of the main application would look like this:

```
package untitled3;

import javax.swing.UIManager;

public class Application1 {
  boolean packFrame = false;

//Construct the application
  public Application1() {
    Frame1 frame = new Frame1();
    //Validate frames that have preset sizes
    //Pack frames that have useful preferred size info, e.g. from their layout

    // This is the line of code that you add
    frame.setModule(new untitled3.CountryDataModel.entities.CountryDataModelServerModule());
    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();
  }
}
```

In this case the class that implements the *CountryDataModelModule* interface is *CountryDataModelServerModule*, which exists in the *entities* package.

Chapter

16

Database administration tasks

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

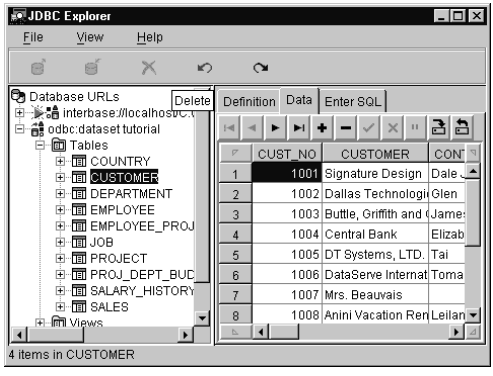
- Exploring database tables and metadata using the JDBC Explorer
- Using the JDBC Explorer for database administration tasks
- Monitoring database connections
- Moving data between databases

Exploring database tables and metadata using the JDBC Explorer

The JDBC Explorer is an all-Java, 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 JDBC Explorer, select Tools | JDBC Explorer from the JBuilder menu.

Figure 16.1 JDBC Explorer as selected from Tools menu



Through a persistent connection to a database, the JDBC Explorer 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 JDBC Explorer window contains a menu, a toolbar, a status bar, 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).

  A plus sign (+) beside an object in the left pane indicates that the object contains other objects below it. To see those objects, click the plus sign. When an object is expanded to show its child objects, the plus sign becomes a minus sign. To hide child objects, click the minus sign.
- 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. Bolded 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 JDBC Explorer by selecting Tools|JDBC Explorer from the menu, then refer to its online help.

Setting up drivers to access remote and local databases

The JDBC Explorer browses databases listed in the Connection URL History List section of the /lib/jbuilder.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 JDBC Explorer 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 plus sign 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 Object | 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 IDEClassPath statement in the JBuilder.INI file.
  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 Object | Delete from the menu to remove the URL.

Note If you're creating a new ODBC URL, you must define its ODBC Data Source though 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. 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 Query (or Run) button, the Next Query button, and the Previous Query button. When an SQL SELECT statement is executed, the results of the query are displayed in an editable grid, which is located below the edit box. This screen may need to be resized to view all its components.

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 an SQL statement.
4. Enter (or paste) an SQL statement in the edit box. If you enter non-SELECT statements, the statement is executed, but no result set is returned.
5. Click the Run 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.

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 grid populated with the data from the selected table. A navigator control is displayed across the top of the grid for navigation and data modification.

You can use the JDBC Explorer 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 grid of all data in the table.
  3. Use the navigator buttons at the top of the grid 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 grid.
  3. To post the edits to the database, select a different record in the grid, or click the navigator's Post button.
  4. To cancel an edit before moving to another record, click the navigator's Cancel button or press *Esc*.

- Insert a new record
  1. Place the cursor on the row before which you wish to insert another row.
  2. Click the navigator'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 database, select a different record in the grid, or click the navigator's Post button.
  5. To cancel an insert before moving to another record, click the navigator's Cancel button or press *Esc*.
- Delete a record
  1. Place the cursor on the row you wish to delete.
  2. Click the navigator's Delete button.

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

- Click the Post button on the navigator.

Using the JDBC Explorer 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 JDBC Explorer.

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, or refer to the documentation for your specific SQL dialect.

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 | JDBC Explorer.
2. From the JDBC Explorer, 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 the JDBC Explorer 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. For example, to create a table named mytable on the data source to which you are connected,

1. Click the Enter SQL tab in the JDBC Explorer.
2. Enter the following in the text area:

```
create table mytable (
lastName char(20),
firstName char(20),
salary numeric(10,2)  )
```

3. Click the Execute Query button.

These steps create an empty table which can be used in a query. Use the JDBC Explorer 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 JDBC Explorer (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 JDBC Explorer if the table permits write access, and if Request Live Queries is checked in

the View | Options dialog box. The Data page displays a grid populated with the data from the selected table.

1. Follow the steps for "Creating the SQL data source" on page 16-5.
2. Select the table you just created in the left window, then select the Data tab in the right window. A grid populated with the data from the selected table displays in the right pane. A navigator control is displayed across the top of the grid for navigation and data modification.

You can now use the JDBC Explorer 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. In the left pane, select a table to view.
  2. Click the Data page tab in the right pane to view a scrollable grid of all data in the table.
  3. Use the navigator buttons at the top of the grid to scroll from record to record.
- Edit a record
  1. Make sure Request Live Queries was checked in the View | Options dialog box before the database was opened.
  2. Edit the record's fields in the grid.
  3. To post the edits to a local cache, select a different record in the grid, or click the navigator's Post button.
  4. To post the edits to the database, click the navigator's Save button.
  5. To cancel an edit before moving to another record, click the navigator's Cancel button or press *Esc*.
- Insert a new record
  1. Place the cursor on the row before which you want to insert another row.
  2. Click the navigator'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 database, select a different record in the grid, or click the navigator's Post button.
  5. To cancel an insert before moving to another record, click the navigator's Cancel button or press *Esc*.
- Delete a record
  1. Place the cursor on the row you want to delete.
  2. Click the navigator's Delete button.

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 16-5, 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 JDBC Explorer.

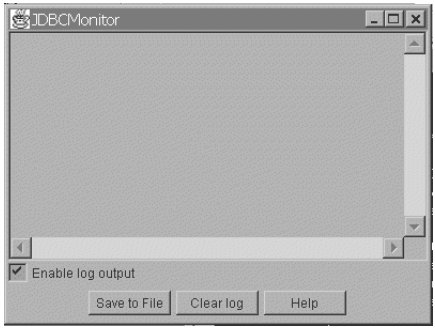
Monitoring database connections

JBuilder provides a JDBC monitoring class which can monitor or manipulate 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 16-9.

JDBC Monitor will monitor any JDBC driver (i.e., any subclass of *java.sql.Driver*) while they are 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 checkbox to enable/disable log output.
- 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. Right-click on the Component Palette, select Properties to bring up the Palette Properties dialog.
2. Select the Data Express page on the Pages page.
3. Select the Add From Archive page.
4. Enter or browse to the file \lib\jbcl3.0.jar in the Select A JAR Or ZIP File box.
5. Select *com.borland.jbcl.sql.monitor* in the Packages Found In Archive list.
6. Select *com.borland.jbcl.sql.monitor.MonitorButton* in the JavaBeans Found In Package list.
7. Click the Install button to add this component to the palette.

You could instead edit the file \bin\palette.ini. Add the following line to the section [Palette_Page3] (the Data Express page):

```
lib\jbcl3.0.jar,com.borland.jbcl.sql.monitor.MonitorButton=*
```

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
outputEnabled	Turns Driver trace on/off.
maxLogSize	Maximum trace log size. Default is 8K.

Moving data between databases

You can use the Data Migration Wizard to migrate, or move, data between desktop database tables and SQL database servers. The wizard moves the actual data, as well as the database schema, from one database to another. Both source and target databases can be either desktop database tables or SQL tables. The Data Migration Wizard only works with databases for which SQL Links are available. You must have the Borland Database Engine (BDE) and the appropriate SQL Links installed.

To use the Data Migration Wizard,

1. Create an alias for both the source and target databases using the Borland Database Engine (BDE) Configuration Utility. Follow the directions for creating an alias. Select the driver for the source or target database, and set all the parameters. For information about creating aliases, refer to the BDE Configuration Utility online help files.
2. Select Wizard | Data Migration Wizard from the menu. In the first screen of the Data Migration Wizard, select the alias or a directory for the source database for the data to be moved from. Desktop databases allow you to select either an alias or a directory; however, SQL server databases always require an alias and may require a login.
3. Select the alias for the target database for the data to be moved to in the second screen of the Data Migration Wizard.
4. Select the tables that you want to move from the source database to the target database by moving them from the Available Tables list to the Selected Tables list.
5. Modify any data types, indexes, or referential integrity that are not supported on the target database.
6. Upsize the data.
7. View the final status report to determine the sequence in which data objects have been upsized, what data objects were upsized, and how they appear on the target. You can now update or modify the data directly on the target.

Before you move any data, you must understand how the data was created on the source database and how it will appear on the target database. Depending upon the structure of your data on the source database and the database objects that are supported on the target database, some modifications may be required. For example, some servers do not allow spaces in field names, some do not allow words like "Date" to be a field name. Referential integrity requires that both tables in the referential integrity relationship be moved to the target database.

Note When moving data to a Microsoft SQL Server, be sure to set the configuration parameters MAX QUERY TIME and TIMEOUT to 20. This ensures data movement.

Note When moving data to a Paradox table, any indexes from the source relation will not be transferred. This is due to the design of Paradox tables, requiring a primary index prior to creating secondary indexes.

Chapter

17

Sample database application

This chapter describes a sample database application developed using Data Express 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\com\borland\samples\dx\dbsample directory of your JBuilder installation under the project name dbSample.jpr.

This application demonstrates the following functionality:

- Connects to the InterBase sample table, employee.gdb, using the *Database* and *QueryDataSet* components. (See Chapter 4, "Connecting to a database" and "Querying a database" on page 5-13.)
- Includes a custom menu that offers the same functionality as custom buttons on the application UI. The menus in this application were created using the Menu Builder. For information on creating menus with the Menu Builder, see "Designing Menus" in the online document *Building Applications with JBuilder*.
- Contains a *GridControl* which displays the data while also demonstrating the following features:
  - Persistent columns, which are columns where structure information typically obtained from the server is specified as a column property instead. This offers performance benefits as well as persistence of column-level properties. (See "Persistent columns" on page 5-46 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 *GridControl* using display masks (the HIRE_DATE column). (See "Adding an Edit or Display Pattern for data formatting" on page 12-3.)

- Data editing that is controlled using edit masks (the HIRE_DATE column). (See "Adding an Edit or Display Pattern for data formatting" on page 12-3.)
- 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 12-9.)

- Includes a *StatusBar* control that displays navigation information, data validation messages, and so on. Messages are written to the *StatusBar* control when appropriate, or when instructed programmatically. (See "Displaying status information" on page 13-4.)
- Displays a *NavigatorControl* when the application loads. Use this control to navigate through the data displayed in the grid. The *NavigatorControl* appears at the same location as the *LocatorControl*. You toggle which control displays by selecting it from the File menu or by selecting the radio button for Navigator or Find.
- Lets you locate data interactively using a *LocatorControl*. (See "Locating data with the LocatorControl" on page 11-11) The *LocatorControl* is hidden initially and is displayed by clicking the Find button or selecting File | Find from the menu.

  The *LocatorControl* has an associated *TextFieldControl* called "Find" that acts as the "label" for the *LocatorControl*. It is displayed when the Find button is selected.

  Clicking the Find button causes the following:

  - The *LocatorControl* is set to *visible* by toggling its *visible* property, and the user types in the EMP_NO value to search for. The *LocatorControl* watches for *Enter* to signal the end of the search criterion.
  - The *LocatorControl* performs the search and results are displayed by the *StatusBar* component.

  The *LocatorControl* disappears (its *visible* property set to **false**) when the Navigator button is selected.
- Resolves changes made to the data in the *QueryDataSet* by using default resolver behavior. (See "Saving changes from a QueryDataSet" on page 6-2.) The Save button of the *NavigatorControl* performs the save. Messages regarding the resolve process are displayed in the *StatusBar* control.

Note Because of data constraints on the EMPLOYEE table, the EMP_NO field cannot be successfully edited. Also, calculated columns are read-only by definition. Data entered into the COUNTRY column must have a matching value in the COUNTRY table.

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 IntlDemo.jpr file in the samples\com\borland\samples\intl directory of your JBuilder installation.

Chapter

18

Database development Q&A

This chapter is comprised of answers to questions posted on the JBuilder Database newsgroup. This document will be posted on the newsgroup, and updated there periodically. To access the Database newsgroup, borland.public.jbuilder.database, point your browser to http://www.borland.com/newsgroups/.

Answers to newsgroup questions

What are the key benefits of DataExpress components?

DataExpress components are modular, allowing for the separation of key functionality. Their design allows them to handle a broad variety of applications. DataExpress components are modular in the following ways:

- 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, and so on.

- 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 and resolver. 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. There are two provider/resolver 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. In the future, borland.com and third parties can create custom provider/resolver component implementations for EJB, application servers, SAP, BAAN, IMS, CICS, and so on.

- Plugable 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 a pure 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 a powerful programmatic interface as well as support for direct data binding to data-aware components such as grid, list, field, navigation via point and click property settings made in a visual designer. JBuilder ships with Java JFC-based visual components that bind directly to DataSet components.

Benefits of the modular DataExpress Architecture

- Network computing

  As mentioned, the provider/resolver approach isolates interactions with arbitrary data sources to two clean points. There are two 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 reestablished 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 *DataSet*s, arbitrary files, and Java Objects. This allows a complete application state to be persisted in a single file storage. DataSets 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.

How are JDBC types mapped to DataSet Column types?

The JDBC to DataSet type mapping is pretty much the same as the JDBC to Java type mapping:

JDBC Types	DataSet Types
java.sql.Types.LONGVARCHAR java.sql.Types.CHAR java.sql.Types.VARCHAR –8: // Oracle ROWID is a String:	Variant.STRING
java.sql.Types.NUMERIC java.sql.Types.DECIMAL	Variant.BIGDECIMAL
java.sql.Types.BIT	Variant.BOOLEAN
java.sql.Types.TINYINT	Variant.BYTE
java.sql.Types.SMALLINT	Variant.SHORT
java.sql.Types.INTEGER	Variant.INT
java.sql.Types.BIGINT	Variant.LONG
java.sql.Types.REAL	Variant.FLOAT
java.sql.Types.FLOAT java.sql.Types.DOUBLE	Variant.DOUBLE
java.sql.Types.VARBINARY java.sql.Types.LONGVARBINARY java.sql.Types.BINARY	Variant.INPUTSTREAM
java.sql.Types.DATE	Variant.DATE
java.sql.Types.TIME	Variant.TIME
java.sql.Types.TIMESTAMP	Variant.TIMESTAMP
java.sql.Types.OTHER	Variant.OBJECT

For a JDBC driver to function in JBuilder, what are the minimum set of JDBC methods it must implement?

JBuilder essentially uses all or most of the methods in the JDBC 1.0 specification. However the DatabaseMetaData interface is very large. These are the methods used in it:

- getBestRowIdentifier
- getColumns
- getDatabaseProductName
- getDriverName

- getIdentifierQuoteString
- getIndexInfo
- getMaxStatements
- getPrimaryKeys
- getProcedureColumns
- getProcedures
- getTables
- getURL
- getUserName
- storesUpperCaseIdentifiers
- storesLowerCaseIdentifiers
- storesMixedCaseIdentifiers(*)
- storesUpperCaseQuotedIdentifiers
- storesLowerCaseQuotedIdentifiers
- storesMixedCaseQuotedIdentifiers(*)
- supportsTransactionIsolationLevel
- supportsTransactions

The (*) marked methods are not currently used, because most drivers implement them incorrectly.

How can DataExpress components (for example, DataSet) be serialized?

Since JBuilder 2.0, all components implement java.io.Serializable. Note that DataSets serialize component state but not data. To serialize the data of a DataSet, use the DataSetData class which has load and extract methods for loading and extracting data to and from a DataSet.

Is there a sample application using serializable DataSets with RMI?

Since JBuilder 2.0, the DataSetData class can extract edited or unedited rows from a DataSet. *DataSetData* implements *java.io.Serialization*, so it can be used as a parameter to an RMI method or manually streamed between a client and server as a stream of bytes. DataSetData also has methods for loading its contents back into a *StorageDataSet*. See these examples:

- samples\com\borland\samples\dx
- samples\com\borland\samples\dx\ntier

Is there a sample Provider and Resolver for a DataSet?

See the following:

- samples\com\borland\samples\dx\providers
- samples\com\borland\samples\dx\datasetdata
- samples\borland\samples\dx\ntier

Are there any large reference applications that use DataExpress components?

The International Online Store makes extensive use of DataExpress and JBCL. See the following:

- samples\borland\samples\intl
- samples\borland\ref*

How can an application set Column values to null in a DataSet?

Call *setAssignedNull*() or *setUnassignedNull*(). These methods are available from any class that extends from ReadWriteRow including all DataSet classes.

How can an application add Columns in a DataSet that will match the contents of my text data file?

- If a file is saved by a TextDataFile, a *.SCHEMA* file is created by the save process. A schema file is a text file that describes the structure of the columns in a DataSet. When the TextDataFile component is used to load the text data file, the schema file is automatically used to initialize columns to a DataSet.
- If there is no *.SCHEMA* file, you can use the StorageDataSet addColumn() or setColumns() methods to initialize the structure of the DataSet before calling the TextDataFile load method. You can use the Column Designer inside JBuilder to generate the setColumns method.

What methods should be used for programmatically adding rows to a DataSet?

- DataSet.addRow() is the simplest way to add a row of data to a DataSet. This method will immediately post a DataRow into a DataSet. This method is more efficient that DataSet.insertRow() described below. New rows are posted at the end of the DataSet unless the DataSet.Sort property is set.
- DataSet.insertRow() allows an application to insert an unposted row before or after the row position that the DataSet is currently posted at. The data-aware visual components use this method to insert a row for editing. When the row is posted, it is posted at the end of the DataSet unless the DataSet.Sort property is set.
- StorageDataSet.loadRow() is a high-speed data-loading method typically used by providers. This mechanism also allows for specifying the insert/delete/update status of a row as it is added. See StorageDataSet.startLoading() and StorageDataSet.endLoading().

How can an application keep inserted/deleted/updated rows from being marked resolved when a saveChanges operation fails?

Since JBuilder 2.0, StorageDataSets with the StorageDataSet.Resolvable property set to true keep track of all inserted/deleted/updated rows in three separate DataSet views. The RowStatus of a row indicates whether it is inserted/deleted/updated. Currently the status of a row can always be read by calling DataSet.getRowStatus().

Inserted/deleted/updated rows tracked by a DataSet are saved back to the data source via a Resolver component. By default the JDBC resolvers provided for DataExpress call DataSet.resetPendingStatus(false) if one or more rows cannot be saved back to the JDBC driver and they call DataSet.resetPending(true) if all operations succeed. Passing true to DataSet.resetPendingStatus() informs the StorageDataSet that the inserted/updated/deleted rows have been resolved and no longer need to be tracked as edited rows. When this happens, deleted rows are forgotten, and inserted/updatedRows have their RowStatus switched to RowStatus.LOADED.

If an application needs finer control over when and if the edited rows are having their RowStatus state reset, there are two things that must be done:

- The Resolver must be instructed not to call DataSet.resetPendingStatus(). For JDBC data sources this can be accomplished by calling the Database. saveChanges(DataSet[] dataSets, boolean doTransactions, boolean postEdits, boolean resetPendingStatus) method with resetPendingStatus set to false.
- When the application knows that the resolution operation should succeed (will not be aborted/rolled back), the application should make the call to DataSet.resetPendingStatus(true). If the operation should be aborted call DataSet.resetPendingStatus(false).

Normally a DataSet.saveChanges() or a Database.saveChanges() (for JDBC-specific data sources) call a DataSet.resetPendingStatus() method with a boolean parameter of false the row status of all updated/inserted/deleted rows as being resolved. For deleted rows, this means forgetting the row ever existed. For inserted/updated rows the RowStatus.INSERTED and RowStatus.UPDATED are replaced with RowStatus.LOADED.

How can an application load a DataSet from a text file when no schema file exists?

Use the Column Designer to add persistent columns to your DataSet that match the number and data types of columns in the text file that needs to be loaded. To open the Column Designer, right-click on the DataSet component and select Activate Designer.

Once the persistent columns have been created, a TextDataFile component can be used to load the text file into the DataSet.

How can an application write a text file from a DataSet?

Check the TextDataFile sample in samples\com\borland\samples\dx\textdatafile\ and examine the usage of the TextDataFile component.

How can the display attributes and values of UI controls be customized based on the values stored in a DataSet?

Since JBuilder 2.0, you can wire the painting event for a DataSet Column component. This event provides complete context of the row/column being painted by a visual control. Using the CustomPaintSite interface that is provided, many display attributes can be updated for the current row/column including color and font. The value that is passed in can also be modified since it is a copy of what is stored in the DataSet.

How can an application avoid closing a DataSet to set Column properties?

The close/open requirement is used mostly for structural changes that other components need notification of. UI controls are the most common components to need these notifications (for example, a column dropped on a DataSet that a GridControl is viewing). Some properties like Column.SetRowId() were added to this category so that the DataStore component could register this change for a persistent DataSet.

Use persistent columns via the Column Designer. This way you can specify rowId before the DataSet is opened the first time.

How can an application prevent UI controls from flickering when a DataSet is closed, refreshed or changed?

If you're doing something that will close a DataSet, for example, changing the query property of a QueryDataSet, write code like the following in between calls to DataSet.enableDataSetEvents(false), and DataSet.enableDataSetEvents(true).

Code sample 18.1 Example

```
queryDataSet.enableDataSetEvents(false);
// Instruct UI control to go into batch mode and not repaint.
queryDataSet.close();
queryDataSet.setQuery(…);
queryDataSet.open();
queryDataSet.enableDataSetEvents(true);  // Instruct UI controls to leave batch mode and
   repaint new contents without flicker.
```

If you're switching the DataSet property for a UI Control, consider using a DataSetView for the control's DataSet property and then setting the StorageDataSet property on the DataSetView when you need to switch to a different DataSet. Once this is done the above steps can be used to avoid flicker when the DataSet for a control is switched.

How can an application generate primary keys?

You can use any of the following methods to generate primary keys:

- Create a lookup table on your SQL server that contains a Table Name column and a Last Key Used column. To allocate a new key value, lock the table, save the current value as the key to use, increment the key value in the lookup table and then release the table lock. The typical way to do this is to start a transaction, read the current value and save it for your new row, issue and update a statement to increment the lookup table's Last Key Used column. This assumes that your connection has a strong enough isolation level to prevent multiple readers. The Intl sample application uses this technique.
- Some DBMS vendors support server-generated values like SQL Servers identity feature. This automatically generates values for primary keys.
- Generate GUIDs or random numbers for primary keys. If there is a conflict (which should be rare), a new value can be generated. GUIDs can combine a variety of information including machine, user, timestamp.
- Use stored procedures when resolving data back. In this case you may employ the lookup technique described above inside a stored procedure. Note that a QueryDataSet can use a ProcedureResolver for saving data back instead of the default QueryResolver. This allows the application freedom in retrieving data, while providing centralized control over the updating of data. The centralized control of the data can help maintain data integrity.

How can I customize the resolving process for primary key generation and propagation to related details?

The first step is to add a Resolver component to your Frame or DataModule container. Currently there's a QueryResolver and a ProcedureResolver for

JDBC-based connections. Set your Query/ProcedureDataSet resolver property to the Resolver component of your choice. The Resolver components have before and after events that can be wired to preprocess and postprocess insert/delete/update operations back to a data source (which may or may not be a JDBC connection). You can use the inserting event to set a primary key value generated using one of the techniques discussed under "How can an application generate primary keys?"

The primary key of a master row can be propagated to its associated details in the Resolver's inserted event.

How can an application debug the retrieval and saving of data to a JDBC data source?

Do one of the following:

- Call `java.sql.DriverManager.setLogStream(System.out)` just before any statements that may invoke JDBC calls. To disable this output, call the same method again with `null` as the parameter instead of `System.out`.
- Use the SQL Monitor via the MonitorButton component. Note that the option settings work only with the DataGateway JDBC drivers.

What is the cause and solution for the "DataSet has no unique row identifiers" error when editing a QueryDataSet?

The JDBC driver in use might not support the JDBC MetaData discovery methods (getPrimaryKeys, getIndexInfo) that DataExpress is using. One solution is to use a manual override of the built-in QueryDataSet MetaData discovery mechanisms. To manually make a query updateable,

1. Set QueryDataSet.MetaDataUpdate property to NONE.
2. Set StorageDataSet.TableName to the name of your SQL table.
3. Set Column.RowId on the columns used as a unique index to the table.
4. If you have a BLOB columns, set their searchable property to false.

Note These properties must be set before the QueryDataSet is opened.

When saving changes to a DataSet, I get an exception of "No rows affected" or "The row specified by the resolution query was not found"

This typically means that the search values in your where clause were not found when executing a searched update or delete query.

Possible causes:

- Columns that are calculated on the server, could cause the comparison in the `where` clause of the update query to fail. Usually this error is encountered only if two successive saveChanges() have been called without an intervening refresh().
- Another user has updated the same row after you retrieved it and before you attempted to save it.
- Column is of an imprecise data type (such as floats and doubles that may incur rounding differences. for example, a floating point comparison failed to match due to rounding differences).

- Column is a fixed-length string that needs space padding. (Some drivers don't pad strings with blanks, which causes failures in comparison. See Database.setUseSpacePadding().)
- Oracle JDBC driver (thin and OCI) has a bug, where a decimal value of 0.0 is reported with a scale of 1 instead of 0. This causes a problem when the value is passed back to the driver as a parameter to an update query. It fails to compare to 0.
- Some JDBC drivers do not handle some international character values in a where clause. Setting the Column.Searchable property to false may work around this problem.

A technique for identifying the cause of the problem:

1. Add a QueryResolver component to your frame, and specify updateMode to "key columns only".
2. Set the resolver property of the problematic queryDataSet to the queryResolver from the above step.
3. Set rowId to true for each column, one at a time, on the queryDataSet.

Warning Do this on test data only, since you might inadvertently change some data that you didn't intend to change.

Possible Solutions:

- Set the searchable property of the column in question to false, so that the column is not included in the where clause of the update query.
- Add a QueryResolver to the application, and set UpdateMode to KEY_COLUMNS or CHANGED_COLUMNS

Cannot save or change BLOB fields?

There are several different behaviors and requirements from various SQL servers for searched updates on BLOB fields like VARCHAR. BDE SQL Links will not include such fields in the where clause of a searched update. We will probably make this same change in the future.

There are three possible solutions for now (the first two are more desirable):

- Set the QueryResolver.UpdateMode property to UpdateMode.KEY_COLUMNS. QueryResolver component must be added to QueryDataSet's container and set on the QueryDataSet's Resolver property.
- Before calling Database.saveChanges(), set the Column.Searchable property to false. This must always be done after the QueryDataSet's query is run (from QueryDataSet.open() or QueryDataSet.executeQuery()) method because the Column.Searchable property will be overridden by MetaData inspection when the query is run.
- Set QueryDataSet.MetaDataUpdate to NONE. This forces you to set all the appropriate MetaData information on the column component including Column.RowId, and Column.Searchable properties.

What are some possible causes of errors when saving changes back to a JDBC data source?

Many JDBC drivers provide cryptic error messages when a request fails. It often helps to view more verbose messages from the driver by calling java.sql.DriverManager.setLogStream(System.out).

Possible causes:

- You have a manually added or Column field to your DataSet. By default calculated columns are treated as not resolvable. For noncalculated columns or calculated columns that have their Resolvable property set to TRUE, saving changes to this DataSet to a JDBC data source may result in error if the column does not exist on the server. To correct this problem, set the resolvable property of these columns to FALSE.
- Your driver does not support prefixing field names with table names (for example, testtable.column1). To correct this problem set useTableName in your Database component to false.
- MetaDataUpdate has been set to NONE, but searchable has not been set to false for BLOB columns. Set the Column.Searchable property to false.

How should the StorageDataSet.MetaDataUpdate property be used?

This property should be used only if a driver has poor metadata support or an application needs the extra performance benefit of avoiding metadata discovery when a query is executed.

If the metaDataUpdate property on a QueryDataSet is set to ALL, then the following properties are controlled by the MetaData information retrieved from the database connection:

- For Column: rowId, precision, scale, searchable
- For QueryDataSet: tableName, schemaName

Note These properties are used to generate various UPDATE, INSERT, and DELETE queries by the QueryResolver component.

Some database drivers currently don't support all the metadata calls, that are needed to determine the above properties correctly. An application can get around this problem by setting metaDataUpdate to NONE and manually setting the properties normally initialized by the built-in metadata discovery mechanisms of QueryDataSet. Some of the more important properties to specify if changes must be saved back are

1. First set metaDataUpdate to NONE.
2. Set tableName on the queryDataSet.
3. If the query contains BLOBs, set the searchable property for the BLOB columns to false.
4. For the column(s) that uniquely identify a row (that is, the primary key), set the rowId property to true.

What is the easiest way to delete the details of a master DataSet?

Since JBuilder 2.0, the easiest way is to set the cascadeDeletes property of the DataSet.MasterLink property to true.

What is the easiest way to change the linking field values for the details of a master DataSet?

Since JBuilder 2.0, the easiest way is to set the cascadeUpdates property of the DataSet.MasterLink property to true.

Why does the number of inserted/deleted/updated rows not change after a call to DataSet.saveChanges() in JBuilder 1?

In JBuilder 1.0 the inserted/deleted/updated rows were not removed from their views after a successful resolve. Instead, their status was just updated with RowStatus.INSERT_RESOLVED, RowStatus.DELETE_RESOLVED or RowStatus.DELETE_RESOLVED.

Since JBuilder 2.0, the successfully inserted/deleted/updated rows are removed from their respective views after a DataSet.saveChanges() or a Database.saveChanges() method call.

Performance tips for retrieving data through a QueryDataSet

Disable MetaData discovery mechanisms for fetch operations. To disable the MetaData discovery mechanisms,

1. Set StorageDataSet.MetaDataUpdate property to MetaDataUpdate.NONE.
2. Set StorageDataSet.TableName property to the table name.
3. Set Column.RowId property for the columns that uniquely and efficiently identify a row.

For small result set queries this can make a big performance improvement. Note that QueryDataSet only performs the metadata discovery operations the first time a query is run.

By default all values from a query execution are loaded into a QueryDataSet on execution. For larger result sets you can

- Set the QueryDataSet.Query property load option to Load.ASYNCHRONOUS. This retrieves the results on a separate thread and allows data to show in a data-aware control immediately.
- Set the QueryDataSet.Query property load option to Load.AS_NEEDED. This retrieves 25 rows at a time by default. New rows are fetched when attempts are made to navigate past the end of the DataSet.
- Set the QueryDataSet.Query property load option to Load.UNCACHED. This is useful for reporting and batch-oriented operations that need to pass the data only once.

For very large result sets you may also consider setting the QueryDataSet.store property to a DataStore component. Using the DataStore as a store mechanism for a DataSet is much more efficient than the default MemoryStore when the size of the

Result set is very large, because the DataStore has data caching mechanisms that can page data in and out of the persistent DataStore.

Download JDBCBenchMark.java from the borland.com Web site to see optimized DataExpress code and a basic comparison of raw JDBC performance and DataExpress components.

Performance tips for saving data through a QueryDataSet

- For deletes and updates, searched updates and deletes are used. The QueryResolver.UpdateMode property setting determines what columns are used in the where clause of these operations. By default a conservative setting of UpdateMode.ALL_COLUMNS is used that includes all searchable (that is, not BLOBs) columns.
- For each call to Database.saveChanges() calls are made to disable or enable a JDBC drivers autocommit mode. If your application calls database.saveChanges() with the useTransactions parameter set to false, then these calls will not be made and the transaction will not be committed.
- By disabling the resetPendingStatus flag in the Database.saveChanges() method, further performance benefits can be achieved. With this disabled, DataExpress will not clear the RowStatus state for all inserted/deleted/updated rows. This is desirable only if you will not be calling saveChanges() with new edits on the DataSet without calling refresh first.

Note If transactions are disabled, you must call database.getJDBCConnection().connection.commit().

How can the class file footprint of a DataExpress application be reduced?

The JBuilder Deployment wizard should be able to exclude most unreferenced classes.

*BeanInfo.class files are used only at design time.

If visual components are not used, then the following com.borland.dx.dataset classes can be excluded at deployment time:

- SingletonDataSetManager.class
- VectorDataSetManager.class
- MatrixDataSetManager.class
- DataSetAware.class
- DataSetModel.class

If DataStore is being used for all DataSets, then the MemoryStore classes are not needed. Currently several of these nonpublic classes are obfuscated, but the following public classes can be excluded:

- MemoryData.class
- MemoryStore.class

TextDataFile.class is used only if data is imported from a text file.

Reading 100,000+ rows into a DataSet causes "java.lang.OutOfMemoryError"

This error indicates that the VM ran out of memory. To set the amount of memory for the VM, use the -mx parameter for java.exe.

By default DataSets use memory storage for result sets (MemoryStore). Here are some ways to reduce the number of rows read in:

- Refine the query or stored procedure to select fewer rows.
- Use a Query/Procedure property load option of Load.UNCACHED or Load.AS_NEEDED.

Or, you can bypass memory datasets and use the Persistent DataStore. Using the DataStore as a store mechanism for a DataSet is much more efficient than the default MemoryStore when the size of the Result set is very large since the DataStore has data-caching mechanisms that can page data in and out of the persistent DataStore.

How does the DataStore provide persistence for mobile/offline computing?

Since JBuilder 2.0, DataStore provides high performance data caching and compact persistence for DataExpress DataSets, files and Java Objects. The data store has a hierarchical directory structure that allows a single DataStore to persist multiple DataSets, files and objects.

The DataStore component can be used with any DataSet that extends from StorageDataSet (TableDataSet, QueryDataSet, ProcedureDataSet). In addition, you can use Java serialization support since JBuilder 2.0. The new DataSetData class can extract edited or unedited rows from a DataSet. DataSetData implements java.io.Serialization so it can be used as a parameter to an RMI method or manually streamed between a client and server as a stream of bytes. DataSetData also has methods for loading its contents back into a StorageDataSet.

To use the DataStore,

1. Drop a DataStore component into your DataModule or Frame and set its FileName property.
2. Set the Store property of your QueryDataSet or ProcedureDataset to the DataStore.
3. Set the StoreName property of your QueryDataSet or ProcedureDataset to some meaningful name.

When should I use JBCL components versus dbSwing (JFC) components?

Our main focus for JBCL components is to maintain (and improve) high quality data-aware components (that is, GridControl, ListControl, FieldControl, NavigatorControl, LocatorControl, StatusControl). Although there are Swing equivalents of many of these, the quality of complex swing controls like tables and overall data-binding support is not as mature or functional as those in the JBCL.

For non-data-aware components like buttons and tree controls, it may be more desirable to use the Swing equivalents. Note that the quality of Swing DataExpress data binding will be significantly improved after JBuilder 2.0.

What is the cause and solution for the "No suitable driver" error in design mode?

The "No suitable driver" error in design mode is almost always due to the JDBC driver in use not being on the IDEClassPath in JBuilder\bin\JBuilder.ini. Use Notepad to edit this file when JBuilder is NOT running.

What can I do when the Password dialog box pops up and freezes?

If the Prompt For Password property is set in the Database.Connection property, there are some situations a thread deadlock bug in JDK <= 1.1.6 treelock monitor of Container.java will occur in a method's addNotify() will show.

To work around this hang, try calling database.openConnection() right after setting up the connection descriptor of a database, which requires a logon dialog box.

Detailed explanation of the JDK defect: The addNotify(), which is called by Java whenever a new component is added, is used by JBCL controls to open the dataSets involved, which in turn requires opening of the database connection. The treelock monitor is kept as this point, which causes problems for any modal dialog.

How can I customize the way visual components handle errors for DataSetExceptions?

Search for "errors, handling" in the Help system for DataSetException.handleException() and see the DataExpress white paper at http://www.borland.com/jbuilder/papers/dataexpress. The white paper includes error handling under the "Editing Constraints" section.

What are the benefits of using a QueryDataSet instead of a JDBC ResultSet?

QueryDataSet provides significant additional functionality on top of the JDBC APIs including ResultSet. See "What are the key benefits of DataExpress components."

Are there any performance benchmarks for raw JDBC vs. DataExpress?

Download JDBCBenchMark.java from the JBuilder Support web page (http://www.borland.com/devsupport/jbuilder/) to see optimized DataExpress code and a basic comparison of raw JDBC performance and DataExpress components.

Why is there an 8-second delay for showing live data in the designer after an application using a DataStore terminates?

If the application that terminates does not close any DataStore that is open, any other process must verify that the DataStore is not still open. This takes about 8 seconds.

To avoid this delay, close all DataStores before exiting your application. For applications that use frames, you can trigger this from the windowClosing event. To wire this event,

1. Select the frame (this).
2. Select the event tab.
3. Double-click on windowClosing.

   This opens the source code, with the cursor in the event handler. Put your DataStore closing code here.

How do DataExpress data types map to the DataGateway and JConnect JDBC drivers for Sybase?

Here is the data type key for Sybase, using JBuilder and DataGateway (DGW):

Sybase	JBuilder/DGW	JBuilder/JConnect
binary	Binary_Stream	Binary_Stream
bit	Boolean	Boolean
char	String	String
datetime	timestamp	Timestamp
decimal	double	BigDecimal
float	double	Double
image	Binary_Stream	Binary_Stream
int	int	Int
*money	double	*not supported*
nchar	String	String
numeric	double	BigDecimal*
nvarchar	String	String
real	double	Float
smalldatetime	timestamp	Timestamp
smallint	short	Short
*smallmoney	double	*not supported*
text	String	String
time_stamp	Binary_Stream	Binary_Stream
tinyint	short	Byte
varbinary	Binary_Stream	Binary_Stream
varchar	String	String

This should help you if you are trying to develop database applications with different implementations and drivers. This can cause some headaches if you don't know about them.

Watch out for numeric data types in Sybase; things can get real interesting with them. Also, realize that you will have to set default values for JConnect or you will receive errors with null values.

What causes the "Attempt to use a DataRow with a DataSet that it is out of sync with" exception?

The error means that the DataRow may be structurally out of date with the DataSet. If the DataRow is instantiated before a structural change is made (like a column being added/deleted/changed/moved), the DataRow will be assumed to be structurally different from the DataSet.

Can the DataSetData component be used to transfer data between two separate instances of a DataSet?

Yes. DataSetData supports extract and load operations that can extract data from a DataSet and load data back into another DataSet. Since the DataSetData implements java.io.Serializable, it can be used to move the data of a DataSet from one machine to another via RMI, CORBA, and so on. If the structure (type and number of Columns) is the same, you can extract from any DataSet and load back into any other separate DataSet. If the structure is not the same, an attempt is made to restructure the DataSet to at least have the same columns as the source DataSet had. The DataSetData.load() operation preserves RowStatus.LOADED/UPDATED/INSERTED/DELETED. It does not provide "synchronize" functionality that could be used to merge changes from one DataSet into another DataSet. It "adds" changes from one DataSet to another DataSet which is necessary for saving changes back to a data source like JDBC.

How can a query or procedure property be changed without losing the data already present in a QueryDataSet or ProcedureDataSet?

If an application has two separate queries or procedures that produce the same number, type, and order of columns, the results of each query or procedure can be accumulated.

To keep the QueryDataSet/ProcedureDataSet from emptying when the query or procedure property is changed, set the AccumulateResults property to true. Note that to change the query or procedure property the DataSet must be closed and then reopened.

How can MS SQL Server 6.5 identity (auto-increment) fields be used with QueryDataSets?

Here's one approach: Set the identity index in the QueryDataSet to NOT resolvable.

For master-detail, an identity field value can be propagated to the details with the following technique:

1. Set the MasterLink CascadeUpdates property of your detail DataSet to true. CascadeUpdates is a feature added in JBuilder 2.0.
2. Drop a QueryResolver component into the same container as your master QueryDataSet and set the QueryDataSet resolver property with it.
3. Provide code with the following logic in your inserted event handlers of the query resolver:

   In the inserted event: launch another query that simply executes "`SELECT @@IDENTITY`". This returns the last value that was inserted into the master QueryDataSet. There are some special considerations on this so please read the MS SQL documentation. The CascadeUpdates setting on the MasterLink property should propagate this setting to your details.

Note The latest release version of SQL Server 6.5 resets identity values on a restart of SQL server. What needs to be done is to create a stored procedure in the master database that runs automatically on SQL Server restart (there's a stored procedure to set this). Your stored procedure should then call another stored procedure in your database

that runs DBCC CHECKIDENT (table name) on each of the tables that have an identity value. Although this has nothing to do with JBuilder and should be fixed in SQL Server 7.0, many people complain about this and don't know why it happens or how to circumvent it.

How can an application convert to a common data type representation when retrieving data from more than one JDBC driver or SQL server?

SQL servers and drivers have various choices for data type representations. Especially for numeric data types. For example an integer data type in an Interbase database would be represented as a BigDecimal in an Oracle database.

DataExpress does not provide a declarative way to deal with this yet, but here is what an application can do in the meantime:

```
Variant value = new Variant;  // can be done once and reused.
getVariant("NumberField", value);
int intValue = value.getAsInt();  // this will do type coercion.
```

How can an application get the designer to recognize persistent columns?

There is a defect that was fixed after JBuilder 2.0 that keeps the designer from seeing persistent columns for a component that extends from a DataSet and has a jbInit(). There is a workaround, which is to add an instance variable for your application's DataSet extension and set it to "this". Here is an example:

```
public class Customer extends TableDataSet {
   Customer customer = this; // This line will enable the designer to see the persistent
                                                                                columns.
   public Customer() {
      super();
      try {
         jbInit();
      }
      catch (Exception e) {
       e.printStackTrace();
      }
   }
   private void jbInit() throws Exception {
   }
}
```

Why does the visual component designer have problems with extensions of the Column component?

Make sure that all Column component class extensions are public. If they are not, the designer cannot instantiate them (this is not a valid Java bean). This leads to all sorts of disfunctionality, including the Picklist property fails, or the StorageDataSet.setColumns() fails or shows blank columns.

How can an application discover all ODBC data sources available to the JDBC ODBC bridge?

com.borland.dx.sql.dataset.RuntimeMetaData.getURLs() will return an enumeration of ODBC data sources registered on the machine. Note that this relies on native method support, and the JBMetaData.dll file must be deployed with an application that uses this method. JBMetaData.dll can be found in the JBuilder bin directory.

Where do the extra ROWID columns come from inside a QueryDataSet using Oracle JDBC driver, and how can they be hidden?

Every Oracle table has a pseudo column with the column name: ROWID. These columns are designed to uniquely identify a certain row in a table. If MetadataUpdate.ROWID is turned on (the default) JBuilder will look for such columns in order to identify the rows you change in order to save the changes back to the Oracle database.

The Oracle JDBC driver has changed so that it reports the existence of such columns such that JBuilder can take advantage of them. Also JBuilder will add these if it detects that you are connecting to an Oracle driver, and will try to use the ROWID column. However if some of the columns you select make up an unique index, JBuilder will not modify your query to add the ROWID column.

JBuilder needs one or more columns to uniquely identify a row on the server in order to make updates, inserts and deletes. JBuilder will generate UPDATE, INSERT, and DELETE queries with a WHERE clause. However if the WHERE clause doesn't contain a column that uniquely identifies a row in your table on the server, it might update several rows, that is, the wanted update and some updates that are unwanted.

That is why JBuilder requires at least one unique rowId in the select query.

If changes will be saved back to the JDBC driver, keep the rowId column resolvable property set to true.

By default, visual controls (like the GridControl) will not display these columns.

How can I ensure that my application makes only one connection to a database server?

It's best to put Database components in a DataModule. A Database component inside one DataModule can be used by QueryDataSets and ProcedureDataSets in the same DataModule or outside the DataModule.

Note that static references to a Database component or its container (that is, DataModule) can cause the Database component to be closed automatically during a Java VM garbage collection. This is because the Java VM will perform "class" garbage collection for classes that are no longer referenced. This phenomenon can be mind boggling, so to ensure your connection stays open, make sure there is always a reference from your application to the DataModule that contains your Database connection.

Can an instance of a QueryResolver component be used by multiple DataSets?

In JBuilder 1.0 and 1.01 this did not work properly if multiple DataSets used the same QueryResolver instance and these same DataSets were being resolved in the same transaction (that is, passed into the same call to Database.saveChanges(DataSets[]).

The problem was corrected in JBuilder 2.0.

For JBuilder 1.0 and 1.01 QueryResolver ResolverEvent logic can be shared by creating a subclass of QueryResolver that provides ResolverEvent logic and then using separate instances of the extended QueryResolver component for each DataSet.

How can an application retrieve an OUT parameter value from an Oracle stored procedure that also returns a ResultSet?

The ResultSet must be discarded before any of the output vars can be returned.

The following is an example of how to do this via the JDBC API:

```
String aCS = "{call multi_receiving(?,?,?)}";
java.sql.CallableStatement cs = database1.getJdbcConnection().prepareCall(aCS);
cs.setInt(1, PurchaseOrdersPR.getInt(0));  // Setup IN parameter
cs.registerOutParameter(2, Types.INTEGER);   // Setup OUT parameter
cs.registerOutParameter(3, Types.INTEGER);   // Setup OUT parameter
cs.execute();
discardResults(cs);   // dump result counts and result sets
com.borland.jbcl.util.Diagnostic.out.println("multi_receiving Passed");
com.borland.jbcl.util.Diagnostic.out.println("Procedure #inserts is:" + cs.getInt(2));
com.borland.jbcl.util.Diagnostic.out.println("Procedure #deletes is:" + cs.getInt(3));
cs.close();
cs = null;
...
static public void discardResults(java.sql.Statement statement)
throws SQLException
{
while (!statement.getMoreResults()&& statement.getUpdateCount() == -1) {}
}
```

How can an application determine if a column value for a row has been updated?

This is useful for visual presentation of edited values. (See Column.painting event.) Currently there is not a built-in method for this, but with some work, the following approach could be used.

At initialization time, set up the following:

```
DataRow originalRow = new DataRow(dataSet);
DataSetView updatedView = new DataSetView();
dataSet.getUpdatedRows(updatedView);
Variant value1 = new Variant();
Variant value2 = new Variant();
```

Then, when you need to see if a column value is changed, do this:

```
If ((dataSet.getStatus() & RowStatus.UPDATED)
!= 0) {
   updatedView.goToInternalRow(dataSet.getInternalRow());
   dataSet.getOriginalRow(updatedView, originalRow);
   originalRow.getVariant(ordinal, value1);
   dataSet.getVariant(ordinal, value2);
   return !value1.equals(value2);
}
return false;
```

How can an application save data loaded from a TextDataFile to a JDBC data source?

By default, data loaded from a TextDataFile is loaded with a RowStatus.Loaded status. Calling saveChanges() on a QueryDataSet or ProcedureDataSet will have no effect because these rows are not viewed as being inserted. Set the property TextDataFile.setLoadAsInserted(true). This will cause all rows loaded from the TextDataFile to have a RowStatus.INSERTED status, and a saveChanges method call to a QueryDataSet or ProcedureDataSet will insert the rows.

Which JBCL components support edit and display masks?

The GridControl, ListControl, and FieldControl have the best support. Support for dbSwing components will be added after JBuilder 2.0.

Is there an example of setting up a fetch as needed master-detail relationship with a QueryDataSet?

The samples\com\borland\samples\apps\cliffhanger application has a sample of this. The key to this is specifying a where clause in your detail QueryDataSet query property. See the following lines from the Cliffhanger DataModule1.java jbInit() method:

```
custOrderDataSet.setMasterLink(new com.borland.dx.dataset.MasterLinkDescriptor
   (customerDataSet, new String[] {"ID"}, new String[] {"CUSTOMERID"}, true));
      custOrderDataSet.setReadOnly(true);
      custOrderDataSet.setQuery(new com.borland.dx.sql.dataset.QueryDescriptor
        (database1, "SELECT ID, CUSTOMERID, ORDERDATE, STATUS, SHIPDATE FROM ORDERS
        WHERE CUSTOMERID = :ID", null, true, Load.ALL));
```

What causes duplicate details to show?

If a detail has a MasterLink property with FetchAsNeeded set to true and the QueryDataSet has no where clause, duplicates can occur as the master is navigated. This is because the same query is run as each row is navigated. If the query is not scoped to the master row, then there is potential for some detail groups to have duplicates. Note that the where clause must use parameter markers for the masterLinkColumns to link the masterLinkColumns to the detailLinkingColumns.

How can a master detail be made to work when there are no rows in the master?

This will work if the detail and master data sets at least have their respective linking columns specified in the MasterLink property. An application can ensure this by setting up persistent columns for the master and detail DataSets.

This is especially important for a QueryDataSet that has a MasterLink with the fetchAsNeeded property set to true. In this case DataExpress cannot determine what columns are present in the detail because its query cannot be run without parameter values from the master. So if the master has no rows, the query cannot be run. Setting persistent columns for the detail MasterLink linking columns will allow the detail dataset to be opened.

How can Oracle stored procedures be used with ProcedureDataSet?

Stored Procedures work only with Oracle's type-2 and type-4 drivers, and the Oracle server version must be 7.3.4 or 8.0.4 or later.

Note Stored "Functions" work with older versions of Oracle Servers and with DataGateway as well.

Here is an example of a stored procedure in a package:

```
CREATE PACKAGE my_pack is
   type cust_cursor is ref cursor return CUSTOMER%rowtype;
   procedure sp_test ( rc1 in out cust_cursor );
end;
CREATE PACKAGE BODY my_pack IS
   PROCEDURE sp_test (rc1 in out cust_cursor) IS
   BEGIN
      open rc1 for select * from CUSTOMER;
   END sp_test;
END my_pack;
```

The call string for this procedure should be: `"{ call my_pack.sp_test(?) }"`. No parameter row is needed. JBuilder will use the cursor to load the dataset.

Note that the result set you need to load the DataSet with must be the first parameter in the stored procedure argument list. If additional parameters need to be sent or received, this can be accomplished by specifying the ParameterRow in the Procedure property.

What is the update order of calc, lookup, and aggregate columns?

Since JBuilder 2.01, all lookup values are initialized in the row that's passed to the calcFields event handler. Aggregate values are not well defined (may not be set). Calculations on Aggregates should be performed in the calcAggAdd() and calcAggDelete() event handlers.

Note that the DataSet.addRow() and DataSet.updateRow() methods will be modified with the appropriate lookup values for any lookup columns they may contain before the calcFields event is called.

The calcAggAdd and calcAggDelete event handlers are both called after the non-calculated aggregates have been updated for an add or delete operation. An update operation is performed as a delete and then and add operation.

How can the "Mismatch between number of parameter markers in query and in ReadRow" error be avoided with Oracle stored procedures?

There are special circumstances when this error can be erroneously thrown. If this problem appears without good cause using Oracle stored procedures, try the following workaround:

1. Place the database1.setConnection() call before the procedureDataSet1.setProcedure() call in jbInit().
2. Place a database1.openConnection() between the two calls above.

This will give the correct error message when working with Oracle stored procedures.

Why does DataExpress add space padding to an Oracle VARCHAR column?

The Oracle JDBC drivers do not apply space padding to CHAR columns when updates or insert statements are executed. To work around this defect, DataExpress will perform space padding for this driver.

If a VARCHAR is mistakenly space padded, it is because DataExpress thinks this is a CHAR column. This can be caused by the Column.SqlType property setting being lost. To solve this problem, do one of the following:

- Make sure the Column.SqlType property is set to java.sql.types.VARCHAR
- Call "Database.setUseSpacePadding(false);"

Note The newer Oracle JDBC drivers may correct the defect. When the defect is corrected, disabling space padding would be the best solution.

How can a DataSet be made to store a code value, but have visual controls display a description value from another DataSet?

Set the Column.PickList property for the column that stores the code value. The PickList property has a lot of flexibility on which columns are used for lookup, fill-in, and display from a separate PickList DataSet.

Note that the GridControl, ListControl, and FieldControl automatically respond to this property setting using an appropriate choice editor.

How can a column value be modified when the field is posted?

Wire the Column.validate() method and modify the variant that is passed in.

How can I prevent an insert/delete/update operation on a DataSet?

Wire the inserting/deleting/updating events on your StorageDataSet. If your application needs to refuse the edit operation, throw a DataSetException from within the event handler.

For an insert error, use this:

```
throw new ValidationException( ValidationException.INSERT_NOT_ALLOWED, "Insert not allowed",
  null);
```

For update, use this:

```
throw new ValidationException( ValidationException.UPDATE_NOT_ALLOWED, "Update not allowed",
  null);
```

For delete, use this:

```
throw new ValidationException( ValidationException.DELETE_NOT_ALLOWED, "Delete not allowed",
  null);
```

You can modify the string messages to your liking. The good thing about ValidationException (which extends from DataSetException) is that if there are any DataSet StatusEvent listeners, the message of the exception will go to that listener. If a StatusBarControl is wired to the DataSet, then it wires itself as a StatusListener to the DataSet. Once this is in place, the message of the ValidationException goes to the status bar line instead of in a more obtrusive ExceptionDialog.

Note that insert/delete/update operations can also be prevented from being "resolved" from a DataSet to a data source such as JDBC by wiring the inserting/deleting/updating events on your resolver component such as a QueryResolver or ProcedureResolver. In these events an ErrorResponse object is passed in that allows you to skip, abort, or retry the operation. You must first associate a Resolver with a StorageDataSet by setting the StorageDataSet.Resolver property before you can wire these events.

Oracle JDBC drivers: What is the cause and solution for the "no rows affected by ..." error?

Problem description:

After changing some values in a data grid, this error occurs when an attempt is made to save the changes back to the database.

Possible causes:

The table may have some CHAR fields. This driver has a problem saving CHAR fields, where the string value doesn't have the exact length specified by the length of the CHAR field. Best workaround for now: Use only VARCHAR fields.

Oracle JDBC drivers: Why is an error generated when the query statement includes a ';'?

For now this is a problem with the Oracle driver. Remove the ';'. Oracle has been notified of this problem.

Oracle Thin JDBC driver: What are the problems with lowercase table names and lowercase column names on Oracle?

There are three issues when accessing or saving data. The net effect is that data access is most likely to be successful if table and column names are stored in uppercase on the server. This is especially true for an Oracle server.

These are the issues:

- If a lowercase table name or lowercase column name is specified in a query, it is required to be in double quotes (Oracle syntax). Here are two examples:
  - SELECT * FROM lowercasetablename
  - SELECT * FROM “lowercasetablename”

  The difference is that Oracle is treating the first example as case-insensitive and will actually access the uppercase equivalent table name. In the second example, the quoted identifier is treated as case-sensitive, and thus treated as a lowercase identifier.
- SELECT * FROM “lowercasetablename” provides data, but cannot be edited because there are no unique row identifiers.

  JBuilder doesn’t handle the generation of the query correctly, which causes it to fail to find a unique rowId. In addition, the Oracle thin JDBC driver throws an exception when java.sql.MetaData.getIndexInfo() is called to find unique indexes for the table. This problem is reported to Oracle.
- The second issue can be avoided by excluding ROWID from the QueryDataSet.MetaDataUpdate property. If however any of the columns are in lowercase, JBuilder will fail to save changes. The problem is that JBuilder has forced the columns to uppercase. This problem can be avoided by adding persistent columns with lowercase names.

Note The last two issues have been fixed in JBuilder 2.0.

What can cause MasterLink property setting to silently fail?

In JBuilder 2.0, some exceptions can be silently ignored when details are being fetched. One reported cause of this is when the data type of the persistent Column.DataType property does not match the data type being retrieved from a provider data source such as Oracle JDBC driver. In this case the Column.DataType had a property of int while the Oracle data source returned BigDecimal.

One way to debug this is to set your QueryDataSet.Provider property to DebugQueryProvider. When you use this provider, you must set the QueryDescriptor property on the Provider. With the DebugQueryProvider you can get a first peek at the Exception that is silently ignored by the fetchAsNeeded detail. Here is the code for DebugQueryProvider:

```
import com.borland.dx.sql.dataset.*;
import com.borland.dx.dataset.*;
import com.borland.jbcl.util.*;

/**
Debug Version of QueryProvider useful for diagnosing
problems with detail dataSets that have "fetchAsNeeded" enabled.
In JBuilder 2.0, the detail loading mechanism can silently
ignore exceptions. Oftentimes there is a complaint about
an unknown Column (for example, because the query failed),
but the cause is hard to determine because an earlier exception
 was silently ignored.
*/
```

```
public class DebugQueryProvider extends QueryProvider
{
   public void provideData(StorageDataSet dataSet,
boolean toOpen)
      throws DataSetException
  {
      try {
         super.provideData(dataSet, toOpen);
   }
   catch(DataSetException ex) {
      Diagnostic.printStackTrace(ex);
      throw ex;
   }
  }
}
```

How can I debug the "Unknown column name" exception for MasterDetail relationships with fetchAsNeeded enabled?

This is sometimes caused by a failed query or stored procedure exception that is silently ignored. Use the DebugQueryProvider in DB167 to track this problem.

What can I do if BLOB Columns cannot be successfully saved back to a JDBC driver?

By default, DataExpress uses the JDBC PreparedStatement.setObject() method to set BLOB parameters against a JDBC driver. If this does not work with your JDBC driver, try setting the Database component UseSetObjectForStreams property to false. This will cause DataExpress to call the JDBC PreparedStatement.setBinaryStream() method for setting BLOB parameters.

How can ProcedureDataSet be used to retrieve a value instead of a ResultSet from a stored procedure?

The answer isn't so simple, since not all JDBC drivers and databases handle stored procedures the same way. However the following notes might be of help:

- ProcedureDataSets should be used only if a ResultSet is generated by executing the procedure. If no ResultSet is generated, the static method ProcedureProvider.callProcedure() should be used. Here's an example:

```
ParameterRow row = new ParameterRow();
row.addColumn("OUT", Variant.STRING, ParameterType.RETURN);
   // some drivers may require this to be ParameterType.OUT
ProcedureProvider.callProcedure(dbCon, "{? = call GET_ARTICLE_LIST()}", row);
String result = row.getString("OUT");
```

- Similarly, to execute a query that doesn't produce a ResultSet, a QueryDataSet should not be used, but the static method QueryProvider.executeStatement() should be used instead, if the query has parameters. If the query doesn't have parameters, use Database.executeStatement().

- Many users prefer to write JDBC code directly, in which
  - database.executeString() maps to java.sql.Statement
  - QueryProvider.executeStatement() maps to java.sql.PreparedStatement
  - ProcedureProvider.executeStatement() maps to java.sql.CallableStatement

What causes a calcFields event handler to be called for every field?

This should happen only if your DataSet is bound to a visual component (DataSet has one or more DataChangeListeners) and your application is calling DataSet.set* methods after calling DataSet.insertRow(). What is happening is that insertRow opens up an unposted row in the DataSet. As each set method is called, the calcFields event handler is called. This allows a calculation to be kept up-to-date for any change made. The intent was to provide immediate feedback when DataSets are bound to visual controls.

To avoid this behavior you can use DataSet.addRow(DataRow) to programmatically add data to your DataSet. InsertRow simulates interactive editing.

Note Don't update the calculation when the boolean "isPosted" parameter to your calcFields event handler is set to false.

How can non-String DataSet Column values be retrieved in a String representation?

JDBC allows non-string values to be retrieved as String values.

Any class that extends from ReadRow, including DataSet, have format() methods where columns can be specified by ordinal or by name. The format() method will use the display format property of the Column to format String and non-String values. This provides more application control over the presentation of values.

How can the "Operation not allowed on an empty DataSet" DataSetException be prevented?

This exception (error code: DataSetException.HAS_NO_ROWS) occurs for a fetchAsNeeded detail dataSet that is being opened on a master DataSet that has no rows. The problem is that the detail does not know what its columns are if it cannot run the fetchAsNeeded query the first time.

The workaround for this would be to at least make the linking columns of the MasterLink property persistent in the detail. This way the detail can open and perform some minimal checking on the MasterLink property.

Can DataExpress access Sybase unsigned TinyInt values as short or int values?

By default, the Jconnect JDBC driver returns TinyInt values as Java byte values, which are signed. Use the following approach to access this data type as a short or int in DataExpress components:

In JBuilder 2.01, type coercion support was added when providing and resolving data. Previously, if you set the Column.DataType property to short or int, the providers/resolvers would coerce the bytes to short or int. The problem with this would be that you would also get the sign extension.

In version 2.01 Column.coerceToColumn and Column.coerceFromColumn were added to allow for overriding automatic data type coercion. This provides a simple event that you can use to mask the byte values before setting the value to a Column of type int.

Here are the steps:

1. Set the Column.DataType property to something bigger than byte, for example, short.
2. Wire the Column.coerceToColumn event to mask the byte value to a short (that is, from.getByte() & 0xFF).

How long are JDBC resources, like statements, held in Query/ProcedureDataSet and Query/ProcedureProvider?

DataExpress will close most JDBC resources when they are no longer needed, for example, ResultSets from data retrieval and metadata requests. ResultSets for data retrieval can stick around if you use a query load option other than Load.ALL. In these cases the ResultSet will be closed after the last row is retrieved.

Database connection is opened automatically as needed. An application must explicitly close the connection when it is no longer needed.

The only JDBC resource left is JDBC statements allocated for queries and stored procedures. DataExpress will cache these resources if database.isUseStatementCaching() returns true (this is the default setting) and the JDBC connection implementation of DataBaseMetaData.getMaxStatements() returns a value greater than 10.

Currently, cached statements will be closed only if

- A provider-related property changes, including Query/Procedure properties, or a Store property is changed.
- StorageDataSet.closeProvider() method is called.
- Query/ProcedureProvider.close() method is called.

  To call this method you must first drop one of these components into your jbInit() and set your StorageDataSet.Provider property to the provider component.

Simply closing the DataSet will not cause a cached statement to be closed. This is a potential resource leak. To avoid this make sure you perform the last two steps when you no longer want the statement cached or disable statement caching by calling

```
Database.setUseStatementCaching(false);
```

To verify interaction of DataExpress components and your JDBC driver, you can call java.sql.DriverManager.setLogStream().

What metadata does DataSetData persist when extracting from a DataSet?

A subset of the Column component properties are extracted into DataSetData. These include

- Column.ColumnName
- Column.DataType

- Column.Hidden
- Column.Resolvable
- Column.Precision
- Column.Scale

If your application needs more Column or DataSet level metadata, you can create your own component that implements java.io.Serialization and contains an instance of DataSetData and additional fields that contain the additional metadata that your application needs.

What can cause a DataSet refresh() or executeQuery() to show a different number of rows or zero rows from one execution to the next?

Queries or stored procedure property settings that use the Load.ASYNCHRONOUS can cause this behavior. This is because the rows are being retrieved on a separate thread of execution. Calling DataSet.getRowCount() right after the query is started will return the number of rows that the separate row retrieving thread has been able to load so far. This will lead to an inconsistent or 0 DataSet.getRowCount().

The StorageDataSet.load.dataLoaded() event can be wired to receive notification that all data has been retrieved by the separate thread.

You can also call StorageDataSet.closeProvider(true) to cause your thread to wait for all rows to be retrieved.

How can an application control the data type used by DataSets when receiving data from a JDBC data source?

Many applications are built to receive data from various JDBC drivers that may return a column's data with different data types, and scale.

Since JBuilder 2.01, there is support for coercing data received from a JDBC driver to the data type specified in a persistent Column component. The data values are coerced to the Column.DataType property setting when data is retrieved and stored in the DataSet. When changes are saved back to the JDBC data source, the values are coerced back to the data type that the JDBC driver expects for that column.

To make a column persistent, use the visual component designer to view the columns of a DataSet. Set the Column.DataType property to the data type your application needs. This will create a Column component and issue the appropriate StorageDataSet.setColumns() call in your jbInit() method.

If you don't like the automatic type coercion, you can override the data coercion by setting the Column.coerceToColumn and Column.coerceFromColumn events. This gives your application full control over data type coercion.

How can an application prevent metadata inquiries by a QueryDataSet for a read-only view?

Set the StorageDataSet.MetaDataUpdate property to MetaDataUpdate.NONE. Set the StorageDataSet.Resolvable property to false if you are not going to save changes back.

How can an application add multiple listeners to a DataExpress event handler that allows only one listener?

A lot of the DataExpress events are single-cast, cracked events. The Java bean specification does not prohibit this, but encourages a multicast, non-cracked approach.

The single-cast, cracked approach is simpler because you do not need to crack an event object get/set context. It can also be more efficient for events that are fired with high frequency, because event objects do not need to be allocated to package up event context.

As an example of how to solve this problem, lets say you want to make the Column changed and validate events as multicast.

Your application need a middle man multicasting class that implements ColumnChangeListener. You call Column.addColumnChangeListener with the middle man class with the first ColumnChangeListener. Then you add the actual listener to the middle man class. The middle man class receives the changed and validate events because it is wired to the Column component's ColumnChangeListener event. It then dispatches this event to all of its listeners. Note that when the last listener is removed from the middle man, he should remove himself as a listener to the Column by calling Column.removeColumnChangeListener().

The com.borland.jbcl.util.MultiCaster class may be of use for dispatching events to multiple listeners.

What is the difference between providing data with DataSet open() and refresh methods?

open() allows the data of a DataSet to be viewed and edited. If the DataSet has a data Provider with an executeOnOpen property like QueryDataSet (which uses a QueryProvider), then the provider will be asked to provide data when the DataSet is "first" opened. If such a DataSet with an executeOnOpen provider property is closed and then reopened, the provider will not be asked to provide again unless StorageDataSet.empty() has been called while the DataSet was closed.

refresh() unconditionally empties out the current data in the DataSet and asks the provider to provide data.

How can a DataSet use the DataStore embeddable database component to provide persistent storage for the DataSet?

To create a table in a DataStore you can use any component that extends from StorageDataSet including TableDataSet, QueryDataSet, and ProcedureDataSet. See the sample below.

Although this example creates columns for a TableDataSet, StorageDataSets with providers like QueryDataSet and ProcedureDataSets will create columns as needed to retrieve the results from a provide operation. So DataStore is an embeddable database that goes a long way in seamlessly supporting the full semantics of the DataExpress data access components.

Code sample 18.2 Example: Creating a new DataStore with a Person table

```
{
   DataStore dataStore = new DataStore();
   dataStore.setFileName("/app/mydatastore.jds");
   dataStore.create();  // It is now created and open for business.
   TableDataSet dataSet = new TableDataSet();
   Column name = new Column();
   name.setDataType(Variant.STRING);
   name.setColumnName("Name");
   Column address = new Column();
   address.setDataType(Variant.STRING);
   address.setColumnName("Address");
   dataSet.setColumns(new Column[] {name, address});
   dataSet.setStore(dataStore);           // If not set, DataSet defaults to using
                                           //MemoryStore
   dataSet.setStoreName("/app/Person"); // specify the name of the DataSet table in the
                                       //DataStore.
   dataSet.open();
   dataStore.close();  // Will automatically close any DataSet using this DataStore.
}
```

Code sample 18.3 Example: Opening an existing DataStore with a Person table

```
{
   DataStore dataStore = new DataStore();
   dataStore.setFileName("/app/mydatastore.jds");
   dataStore.open();  // Only works if DataStore already exists.
   TableDataSet dataSet    =    new TableDataSet();
   dataSet.setStore(dataStore);
   dataSet.setStoreName("/app/Person"); // specify the name of the DataSet table in the
                                       //DataStore.
   dataSet.open();
}
```

Why does my QueryDataSet show only 50 rows when using a DataStore component?

At design time, the StorageDataSet.MaxDesignTimeRows is set to 50. So if the query was initially run in the designer, you got 50 rows. If you restart your application, the StorageDataSet using the DataStore knows it already has data (the 50 rows from design time), so the provider query does not need to be executed. To force the query to be reexecuted at runtime call QueryDataSet.executeQuery(), or QueryDataSet.refresh().

MemoryStore actually behaves much the same way if you open/close/open a QueryDataSet: it does not execute the query on the second open. The difference is that DataStore remembers its data, so that when you restart the application it does not think it needs to execute the query again, because it already has data.

Why does my application have problems reading InputStreams from a DataStore?

In JBuilder 2.01 there was a known problem with setting an InputStream column with a java.io.BufferedInputStream. The BufferedInputStream did not always return the number of bytes requested even when it was not at the end of the file. This behavior was unanticipated by the InputStream handling logic in DataStore.

The workaround was to avoid BufferedInputStream or any stream that behaves in this fashion. The problem is fixed in this release.

Index

Symbols

Numerics

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W