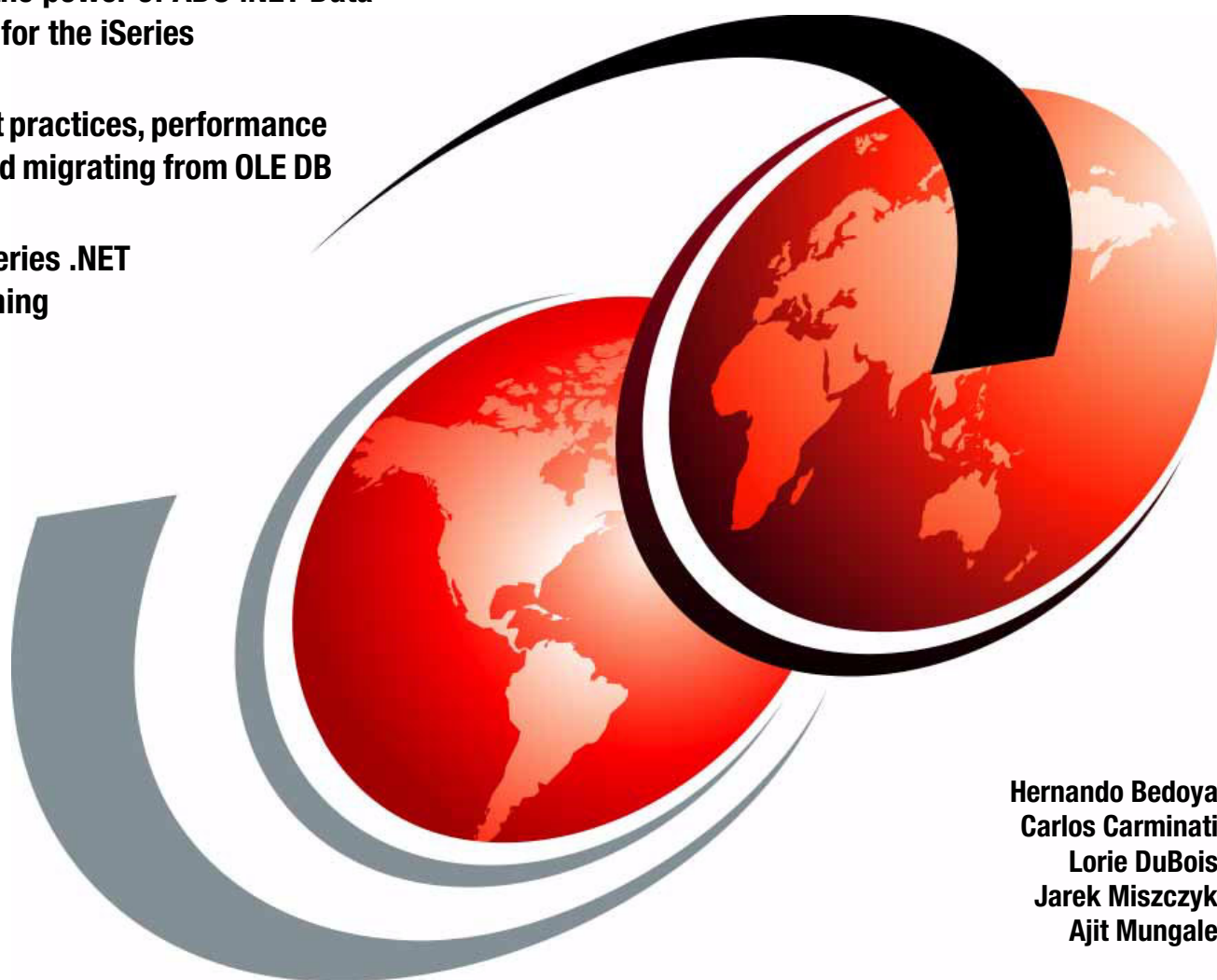


Integrating DB2 Universal Database for iSeries with Microsoft ADO .NET

Discover the power of ADO .NET Data Providers for the iSeries

Learn best practices, performance tuning, and migrating from OLE DB

Master iSeries .NET programming



Hernando Bedoya
Carlos Carminati
Lorie DuBois
Jarek Miszczyk
Ajit Mungale

Redbooks



International Technical Support Organization

**Integrating DB2 Universal Database for iSeries with
Microsoft ADO .NET**

April 2005

Note: Before using this information and the product it supports, read the information in “Notices” on page vii.

First Edition (April 2005)

This edition applies to Version 5, Release 3, Modification 0 of OS/400 (product number 5722-SS1).

© Copyright International Business Machines Corporation 2005. All rights reserved.

Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Notices	vii
Trademarks	viii
Preface	ix
The team that wrote this redbook	ix
Become a published author	x
Comments welcome	xi
Part 1. Background	1
Chapter 1. Introduction to DB2 UDB for iSeries	3
1.1 An integrated relational database	4
1.2 DB2 UDB for iSeries: an overview	4
1.2.1 DB2 UDB for iSeries basics	5
1.3 Connectivity options to DB2 UDB for iSeries	6
1.3.1 Multiplatform connectivity	6
1.3.2 Windows platform connectivity	7
1.4 DB2 UDB for iSeries sample schema	8
Chapter 2. Introduction to the Microsoft .NET framework	11
2.1 Description of .NET	12
2.1.1 Architecture	12
2.1.2 Platform support	13
2.2 Common Language Runtime (CLR)	14
2.2.1 Runtime execution environment	14
2.3 Class libraries	15
Chapter 3. ADO .NET object hierarchy	17
3.1 Overview of ADO .NET	18
3.1.1 Connection	19
3.1.2 Command	20
3.1.3 DataReader	21
3.1.4 DataAdapter	22
3.1.5 DataSet	22
3.2 Connected mode	23
3.3 Disconnected mode	25
Part 2. Providers	31
Chapter 4. IBM DB2 UDB for iSeries .NET provider	33
4.1 Introduction	34
4.2 IBM.Data.DB2.iSeries architecture	34
4.2.1 ADO.NET interfaces	34
4.2.2 Host server jobs	35
4.2.3 Supported features	36
4.2.4 Unsupported features	38
4.3 Before we begin	39
4.3.1 PC setup	39
4.3.2 Host setup	40
4.4 Getting started	41

4.4.1	Displaying the technical reference	41
4.4.2	Starting Visual Studio .NET	42
4.4.3	Adding an assembly reference to the provider	43
4.4.4	Adding a namespace directive	44
4.5	Provider basics	45
4.5.1	A simple connection example	45
4.5.2	iDB2Connection and ConnectionString properties	48
4.5.3	iDB2Command properties and methods	65
4.5.4	Using parameters in your SQL statements	74
4.5.5	Calling stored procedures	79
4.5.6	Choosing your execute method	86
4.5.7	Provider data types	87
4.5.8	Handling exceptions	102
4.6	Common tasks	107
4.6.1	A DataReader example	107
4.6.2	A simple DataAdapter with CommandBuilder example	110
4.6.3	Using transactions	116
4.6.4	Calling a program by wrapping it in a stored procedure	120
4.6.5	Calling a program or CL command using QCMDXEC	120
4.6.6	Choosing between iDB2DataReader and iDB2DataAdapter	127
4.7	Advanced topics	129
4.7.1	Internationalization and support for multiple languages	129
4.7.2	Using large objects (LOBs)	132
4.7.3	Updating DataSets	136
4.7.4	Using iDB2CommandBuilder	139
4.7.5	Using DataLinks	141
4.7.6	Connection pooling	143
4.7.7	Deploying your application	146
4.8	Coding for performance and best practices	146
4.9	Migrating from ADO and OLE DB to ADO.NET	149
4.9.1	ADO objects and how they map to ADO.NET objects	149
4.9.2	ADO recordsets versus ADO.NET DataReaders and DataAdapters	150
4.9.3	Updating tables	150
4.9.4	Mapping OLE DB properties to ADO.NET	151
4.9.5	Examples showing an OLE DB application rewritten to use ADO.NET	152
4.10	Troubleshooting	166
4.10.1	Handle exceptions using try/catch blocks	166
4.10.2	Make sure your server jobs are running	167
4.10.3	Use provider traces via the cwbmptrc utility	167
4.10.4	Enable server-side diagnostics	168
4.10.5	Use communication traces via the cwbcotrc utility	168
4.10.6	Overriding your ConnectionString	168
4.10.7	Gathering information for IBM Support	169
4.11	Writing code for provider independence	171
4.11.1	Writing provider-independent code with ADO.NET 1.0 and 1.1	174
4.11.2	Writing provider-independent code with ADO.NET 2.0	175
Chapter 5.	IBM DB2 for LUW .NET provider	177
5.1	DB2 Connect overview	178
5.2	Installing and configuring DB2 Connect	178
5.2.1	Host server jobs	178
5.2.2	Prerequisites	179
5.2.3	Installation procedure	179

5.2.4 Connecting to an iSeries database	180
5.3 IBM DB2 Development Add-In overview	182
5.3.1 Registering the IBM DB2 Development Add-In	183
5.3.2 Unregistering the IBM DB2 Development Add-In	183
5.3.3 DB2 Toolbar	184
5.3.4 DB2 Database Project type	184
5.3.5 IBM Explorer	186
5.4 IBM DB2 data controls	186
5.5 LUW provider features	194
5.5.1 Classes to implement ADO.NET interfaces	194
5.5.2 Data types	196
5.5.3 Unsupported features	196
5.6 Getting started	197
5.6.1 Starting Visual Studio .NET	197
5.6.2 Displaying the technical reference	198
5.6.3 Adding an assembly reference to the provider	198
5.6.4 Adding a namespace directive	199
5.6.5 Using the DB2Connection object and the ConnectionString	199
5.6.6 Using the DB2Command object	201
5.6.7 Using the DB2DataReader object	202
5.6.8 Using the DB2DataAdapter object	203
5.7 Advanced topics	204
5.7.1 Using large objects (LOBs)	204
5.7.2 Using the DB2CommandBuilder object	207
5.7.3 Performing transactions	209
5.8 Best practices	223
5.8.1 Connection pooling	223
Chapter 6. Selecting the .NET provider	225
6.1 ODBC .NET Data Provider	226
6.2 OLE DB .NET Data Provider	229
6.3 Provider performance	230
6.4 Conclusions	234
Part 3. Scenarios	235
Chapter 7. ASP .NET scenario (Web forms)	237
7.1 An overview of ASP.NET	238
7.1.1 ASP .NET Web page (Web form)	238
7.1.2 How does ASP .NET work?	239
7.1.3 Configuration files in ASP .NET	240
7.2 Web controls	241
7.3 Using the IBM DB2 UDB for iSeries .NET provider	242
7.4 Using the IBM DB2 for LUW .NET provider	249
Appendix A. Sample programs	257
Samples for the IBM DB2 UDB for iSeries .NET provider	258
Sample for the IBM DB2 for LUW .NET provider	259
Appendix B. Additional material	261
Locating the Web material	261
Using the Web material	261
System requirements for downloading the Web material	261
How to use the Web material	262

Related publications	263
IBM Redbooks	263
Other publications	263
Online resources	263
How to get IBM Redbooks	264
Help from IBM	264
Index	265

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing, IBM Corporation, North Castle Drive Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law. INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

AIX®
AS/400®
DB2 Connect™
DB2 Universal Database™
DB2®
DRDA®
@server®
eServer™

i5/OS™
IBM®
ibm.com®
iSeries™
Operating System/400®
OS/390®
OS/400®
PartnerWorld®

Rational®
Redbooks™
Redbooks (logo) ™
WebSphere®
XDE™
z/OS®

The following terms are trademarks of other companies:

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

Preface

Customers have been using the IBM® DB2® UDB for iSeries™ for many years with data access technologies such as ODBC and OLE DB. The newest data access technology from Microsoft® is called ADO.NET. Applications that use ADO.NET with the iSeries can work with several different Microsoft .NET providers:

- ▶ The IBM.Data.DB2.iSeries provider, a .NET-managed provider new to iSeries Access for Windows® in V5R3
- ▶ The IBM.Data.DB2 provider, a .NET provider that works with all IBM @server™ platforms, in conjunction with DB2 Connect™
- ▶ Microsoft System.Data.OleDb provider, as a bridge to one of the OLE DB providers included with iSeries Access for Windows (IBMDA400, IBMDASQL, and IBMDARLA)
- ▶ Microsoft System.Data.Odbc provider, as a bridge to the ODBC driver included with iSeries Access for Windows

This IBM Redbook shows customers how to effectively use ADO.NET to harness the power of DB2 UDB for iSeries, showing examples, best practices, pitfalls, and comparisons between the different ADO.NET data providers.

The team that wrote this redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization, Rochester Center.

Hernando Bedoya is an IT Specialist at the IBM ITSO in Rochester, Minnesota. He writes extensively and teaches IBM classes worldwide in all areas of DB2 UDB for iSeries. Before joining the ITSO more than four years ago, he worked for IBM Colombia as an AS/400® IT Specialist doing pre-sales support for the Andean countries. He has 20 years of experience in the computing field and has taught database classes in Colombian universities. He holds a Masters degree in computer science from EAFIT, Colombia. His areas of expertise are database technology, application development, and data warehousing.

Carlos Carminati is the IT Manager at d2B Network, a firm specializing in graphical design and software development using Internet technologies. His firm has a development center in Uruguay that serves customers in Argentina, Brazil, Colombia, Mexico, and Puerto Rico. He has worked in software development for more than 25 years, using Cobol, C, C++, Java™, and C#. He has important experience with the .NET Framework, architecting solutions that integrate with other systems and platforms.

Lorie DuBois is a Software Engineer at the IBM development lab in Rochester, Minnesota. She owns the development and design of the IBM DB2 UDB for iSeries .NET provider, and has spent the past seven years working on various iSeries Access for Windows technologies, including OLE DB. She holds a degree in Computer Science from the University of Minnesota Institute of Technology and has 20 years of experience in the personal computing field. Her areas of expertise include client database technologies, middleware software development, C#, and C++.

Jarek Miszczyk is a Senior Software Engineer at the Solutions Enablement organization in Rochester, Minnesota. His mission is to provide consulting services to Independent Software

Vendors, IBM customers, and other IBM organizations for issues related to DB2 UDB for iSeries. He also writes extensively and teaches IBM classes in all areas of the iSeries database.

Ajit Mungale is an IT Specialist in IBM and Microsoft MVP for C#. He is the author of several other books about .NET on a wide variety of subjects. He has extensive experience with Microsoft technologies and has worked with almost all languages and technologies. He also has experience with such IBM products as WebSphere®, MQ Series, and DB2. He owns patents in security and other areas.

Thanks to the following people for their contributions to this project:

Thomas Gray
Marvin Kulas
Joanna Pohl-Miszczyk
International Technical Support Organization, Rochester Center

David Dilling
Michael J Swenson
IBM Rochester Support Center

Yvonne Griffin
IBM Rochester Information Development

Brent Nelson
IBM Rochester Development Lab

Kent Milligan
IBM Rochester PartnerWorld®

Fredy Cruz
IBM Colombia - ISV Center

Brent Gross
IBM Toronto Lab

Become a published author

Join us for a two- to six-week residency program! Help write an IBM Redbook dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You will team with IBM technical professionals, Business Partners, and/or customers.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you will develop a network of contacts in IBM development labs, and increase your productivity and marketability.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us because we want our Redbooks™ to be as helpful as possible. Send us your comments about this or other Redbooks in one of the following ways:

- Use the online **Contact us** review redbook form found at:

ibm.com/redbooks

- Send your comments in an e-mail to:

redbook@us.ibm.com

- Mail your comments to:

IBM Corporation, International Technical Support Organization
Dept. JLU Building 107-2
3605 Highway 52N
Rochester, Minnesota 55901-7829



Part 1

Background

In this part we give a short introduction to DB2 Universal Database™ for iSeries. We also introduce the Microsoft .NET framework, specifically the ADO .NET object hierarchy.



Introduction to DB2 UDB for iSeries

This chapter includes:

- ▶ An introduction to DB2 Universal Database for iSeries
- ▶ An overview of the contents in this publication
- ▶ A definition of the sample schema

1.1 An integrated relational database

Integration has been one of the major elements of differentiation of the iSeries server in the information technology marketplace. The advantages and drawbacks of fully integrated systems have been the subject of countless disputes over the past few years. The success of the AS/400 system and iSeries server indicates that integration is still considered one of the premier advantages of this platform. Security, communications, data management, backup, and recovery: All of these vital components have been designed in an integrated way on the AS/400 system and iSeries server. They work according to a common logic with a common end-user interface. They fit together perfectly, because all are part of the same software: the Operating System/400® (OS/400®) and now i5/OS™ on i5.

The integrated relational database manager has always been one of the most significant features of the iSeries server. Relying on a database manager that is integrated into the operating system means that virtually all user data on the iSeries server is stored in a relational database, and that the access to the database is implemented by the operating system itself. Some database functions are implemented at a low level in the iSeries server architecture, and some are even performed by the hardware.

Several years ago a survey pointed out that a significant percentage of iSeries server customers did not even know that all of their business data is stored in a relational database. This might sound strange if you think consider the integrated database as one of the main technological advantages of the iSeries platform. On the other hand, this means that thousands of customers use, manage, back up, and restore a relational database every day without even knowing that they have it installed on their system. This level of transparency has been made possible by the integration and undisputed ease of use of this platform. These have been key elements of the success of the AS/400 and iSeries server database system in the marketplace.

During the past couple of years, each new release of OS/400 has enhanced the DB2 Universal Database for iSeries with a dramatic set of new functions. As a result of these enhancements, the iSeries server has become one of the most functionally rich relational platforms in the industry. Now we have the i5 and a number of connectivity options, as noted in 1.3, “Connectivity options to DB2 UDB for iSeries” on page 6.

DB2 Universal Database for iSeries is a member of the DB2 Universal Database family of products, which includes DB2 UDB for OS/390 and DB2 Universal Database. The DB2 Universal Database family is the IBM solution in the marketplace of relational database systems. It guarantees a high degree of application portability and a sophisticated level of interoperability among the various platforms that are participating in the family.

1.2 DB2 UDB for iSeries: an overview

This section provides a quick overview of the major features of DB2 UDB for iSeries. You can find a full description of the functions that are mentioned in this section in several IBM manuals, such as *Database Programming* and *SQL Reference*. For links to these and other useful documentation, visit:

- ▶ DB2 Information Center
<http://publib.boulder.ibm.com/infocenter/db2help/index.jsp>
- ▶ IBM Publications Center
<http://www.ibm.com/shop/publications/order>

1.2.1 DB2 UDB for iSeries basics

As previously mentioned, the major distinguishing characteristic of the DB2 UDB for iSeries database manager is that it is part of the operating system. In practice, this means that a large majority of your iSeries server data is stored in the relational database. Although the iSeries server also implements other file systems in its design, the relational database on the iSeries server is the one most commonly used by customers. Your relational data is stored in the database, as is typical non-relational information such as the source of your application programs.

Physical files and tables

Data on the iSeries server is stored in objects called *physical files*. Physical files consist of a set of *records* with a predefined layout. Defining the record layout means that you define the data structure of the physical file in terms of the length and the type of *data fields* that participate in that particular layout.

These definitions can be made through the native data definition language of DB2 UDB for iSeries, called *Data Description Specifications* (DDS). If you are familiar with other relational database platforms, you are aware that the most common way to define the structure of a relational database is by using the data definition statements provided by the *Structured Query Language* (SQL). This is also possible on the iSeries server.

Attention: The SQL terminology can be mapped to the native DB2 UDB for iSeries terminology for relational objects, and we use these terms interchangeably in this book:

- ▶ A *SQL table* is equivalent to a DDS-defined physical file.
- ▶ *Table rows* equate to physical file records for DB2 UDB for iSeries.
- ▶ *SQL columns* are synonymous with record fields.

Logical files, SQL views, and SQL indexes

By using DDS, you can define *logical files* on your physical files or tables. Logical files provide a different view of the physical data, enabling column subsetting, record selection, joining multiple database files, and so on. They can also provide physical files with an *access path* when you define a *keyed logical file*. Access paths can be used by application programs to access records directly by key or for ensuring uniqueness.

On the SQL side, there are similar concepts. A *SQL view* is almost equivalent to a native logical file. The selection criteria that you can apply in a SQL view is much more sophisticated than in a native logical file. A *SQL index* provides a keyed access path for the physical data exactly the same way that a keyed logical file does. Still, SQL views and indexes are treated differently from native logical files by DB2 UDB for iSeries, and they cannot be considered to exactly coincide.

Database file refers to any DB2 UDB for iSeries file, such as a logical or physical file, a SQL table, or view. Any database file can be used by applications to access DB2 UDB for iSeries data.

Terminology

Because the DB2 Universal Database for iSeries server evolved from the built-in database present in the AS/400 that was born before SQL was widely-used, OS/400 and i5/OS use different terminology from what SQL uses to refer to database objects. Table 1-1 on page 6 shows the terms and their SQL equivalents. The terms have been interchanged throughout this book.

Table 1-1 SQL terms and OS/400 terms cross-reference

SQL term	iSeries term
Schema	Library, collection, schema
Table	Physical file
View	Non-keyed logical file
Index	Keyed logical file
Column	Field
Row	Record
Log	Journal
Isolation level	Commitment control level

1.3 Connectivity options to DB2 UDB for iSeries

One of the key value propositions for DB2 UDB for iSeries is the large number of database interfaces that are available to customers and programmers designing client/server or multi-tier applications. In this section, we describe the multiplatform connectivity of DB2 UDB for iSeries as well as the Windows client connectivity.

1.3.1 Multiplatform connectivity

Figure 1-1 shows the interfaces that are available from a range of platforms including other iSeries, Linux®, AIX®, and Windows. The next few figures describe some of the enhancements for users of the .NET, JDBC, ODBC, OLE DB, and CLI interfaces.

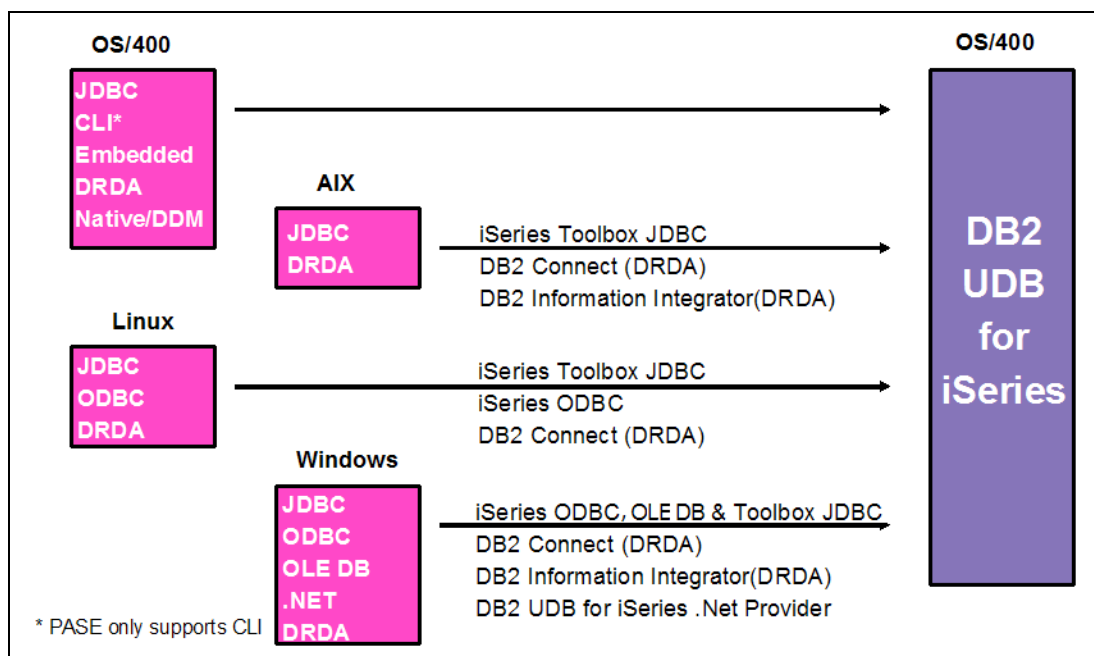


Figure 1-1 Multiplatform connectivity to iSeries

1.3.2 Windows platform connectivity

Currently, there are four providers that can be used to access DB2 UDB for iSeries from .NET applications:

- ▶ The new ADO.NET managed provider for iSeries Access for Windows, IBM.Data.DB2.iSeries, which we discuss in Chapter 4, “IBM DB2 UDB for iSeries .NET provider” on page 33.
- ▶ The DB2 for Linux, UNIX®, and Windows (LUW) managed provider implemented by IBM software group, which we discuss in Chapter 5, “IBM DB2 for LUW .NET provider” on page 177.
- ▶ ODBC: The Microsoft-supplied ODBC bridge provider using the iSeries Access for Windows ODBC driver for underlying database connectivity.
- ▶ OLE DB: The Microsoft-supplied OLEDB bridge provider using one of the iSeries Access for Windows OLE DB providers for underlying database connectivity.

If you are using the .NET framework to connect to an iSeries host, choose the IBM.Data.DB2.iSeries database provider to support your PC and iSeries SQL application development. This managed provider offers better performance than using the System.Data.OleDb provider to bridge to an iSeries Access OLE DB provider, or using the Microsoft.Data.Odbc provider to bridge to the iSeries Access ODBC driver.

The OLE DB .NET Data Provider uses native OLE DB through a COM interop module to enable data access. This provider is a *bridge* that handles calls from .NET into a traditional COM-style OLE DB Provider (IBMDA400, IBMDASQL, or IBMDARLA in the case of an iSeries server). The IBM development lab in Rochester, Minnesota, has recently enhanced these providers to interact with the OLE DB .NET Data Provider to facilitate access to the iSeries database. The OLE DB bridge involves jumping in and out of the .NET Framework environment for every interface call because, from the .NET point of view, OLE DB providers constitute unmanaged code (meaning that it has been compiled directly into a binary executable). Managed code is compiled into a .NET assembly that can be executed in the context of the .NET Common Language Runtime (CLR). In order for managed code to call unmanaged code, marshaling of data must take place, and this can have an impact on overall performance. Therefore, Microsoft recommends that a managed provider be used in the .NET environment.

Similar considerations apply when accessing iSeries through the ODBC .NET Data Provider.

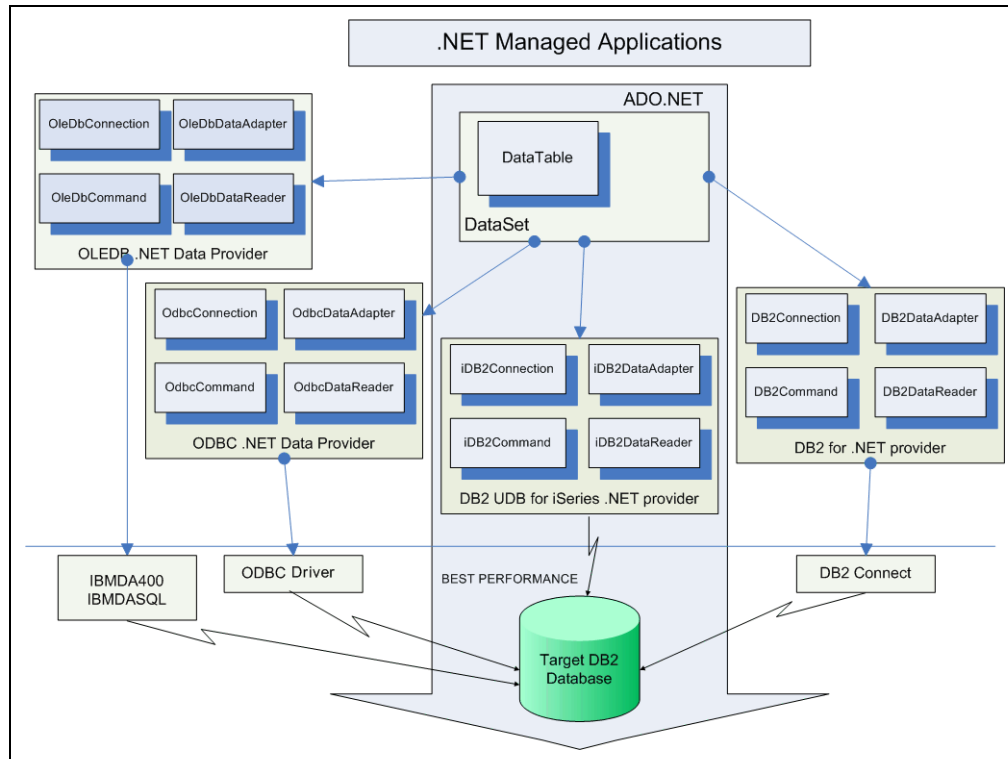


Figure 1-2 ADO.NET providers supporting iSeries

1.4 DB2 UDB for iSeries sample schema

Within the code of OS/400 starting with version V5R1M0, there is a stored procedure that creates a fully functioning database. This database contains tables, indexes, views, aliases, and constraints. It also contains data within these objects.

The database also helps with problem determination because the program is shipped with the operating system code. By calling a simple program, you can create a duplicate of this database on any system running most recent releases. This enables customers and support staff to work on the same database for problem determination.

Throughout many parts of this book, examples are shown that use this sample database. To set up this database, open iSeries Navigator (a part of the IBM iSeries Access for Windows suite) and click **Databases** under the iSeries icon in the left panel. You should see **Databases task** on bottom of this window. Click **Run a SQL Script**. This opens a new window.

Create the database by issuing the following SQL statement:

```
CALL QSYS.CREATE_SQL_SAMPLE('SAMPLEDB')
```

You can find this statement in the example pull-down box of the Run SQL Script window (Figure 1-3 on page 9).

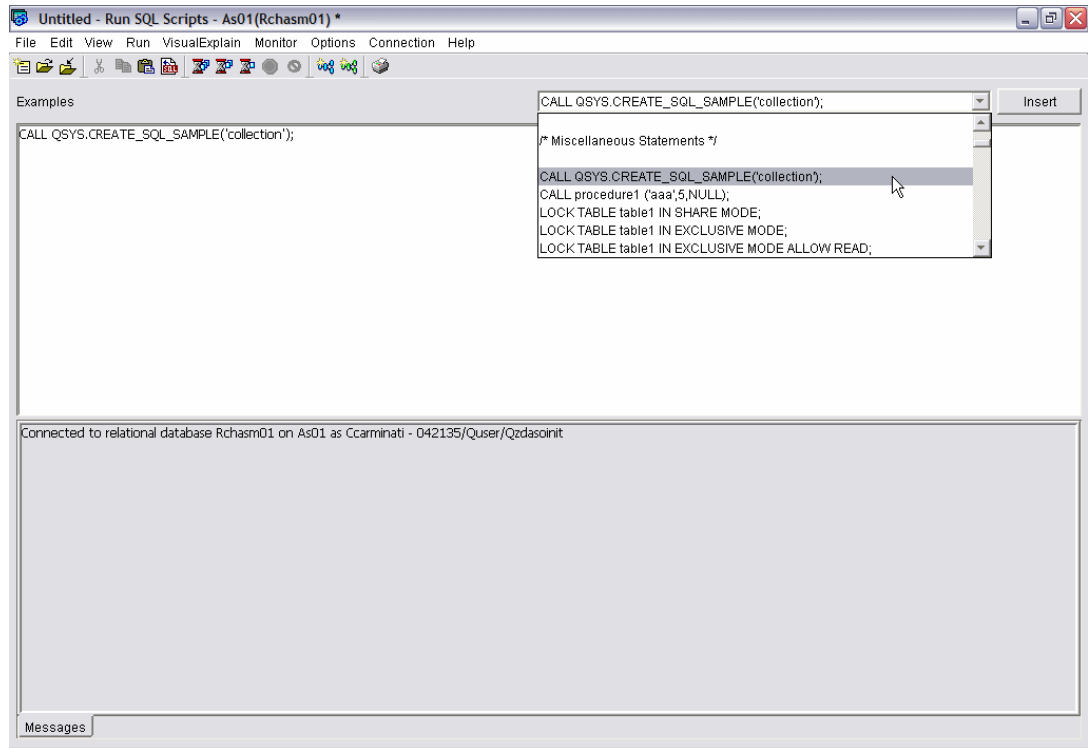


Figure 1-3 Example display showing the schema CREATE statement

Replace **collection** with a valid schema name and click **Run** → **All** from menu or press Ctrl+R. Your sample database is created.

Important: The schema name you type here must be uppercase (all capital letters). This sample schema will be used in future DB2 Universal Database for iSeries documentation.

As a group, the tables include information that describes employees, departments, projects, and activities. This information makes up a sample database demonstrating some of the features of DB2 Universal Database for iSeries. Figure 1-4 on page 10 shows an entity-relationship (ER) diagram of the database created with reverse engineering capabilities of Rational® XDE™ for Visual Studio.

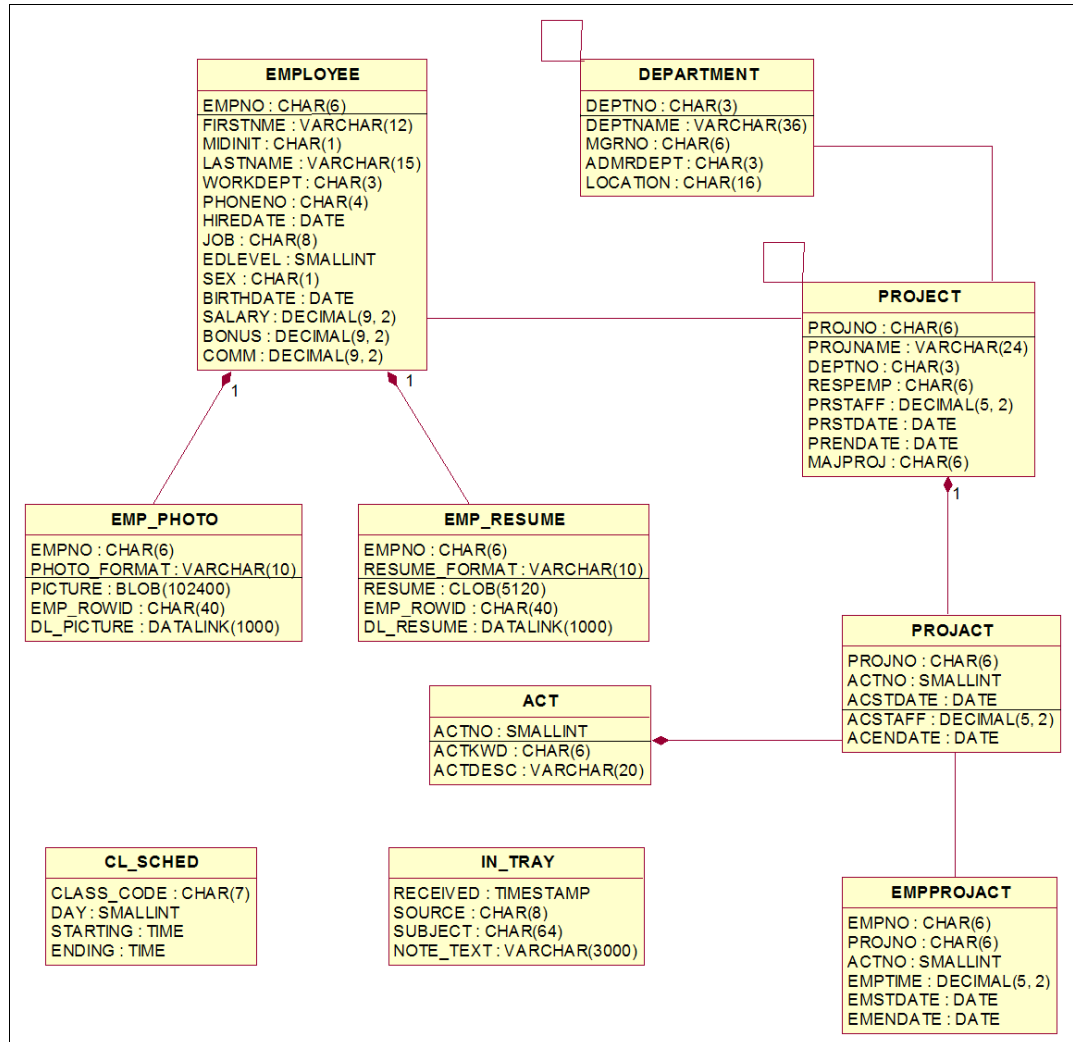


Figure 1-4 Sample schema: ER diagram

The tables are:

- ▶ Department Table (DEPARTMENT)
- ▶ Employee Table (EMPLOYEE)
- ▶ Employee Photo Table (EMP_PHOTO)
- ▶ Employee Resume Table (EMP_RESUME)
- ▶ Employee to Project Activity Table (EMPPROJACT)
- ▶ Project Table (PROJECT)
- ▶ Project Activity Table (PROJACT)
- ▶ Activity Table (ACT)
- ▶ Class Schedule Table (CL_SCHED)
- ▶ In Tray Table (IN_TRAY)

Indexes, aliases, and views are created for many of these tables. The view definitions are not included here. Three other tables are created that are not related to the first set:

- ▶ Organization Table (ORG)
- ▶ Staff Table (STAFF)
- ▶ Sales Table (SALES)

Note: Many of the examples in this book use this SAMPLEDB database.



Introduction to the Microsoft .NET framework

This chapter provides an overview of .NET, the .NET Framework, and capabilities, technologies, and specifications.

This chapter discusses the following items:

- ▶ Description of .NET and its architecture
- ▶ The Common Language Runtime
- ▶ Class libraries shipped with the .NET Framework

2.1 Description of .NET

.NET is a language-neutral environment for writing programs that can easily interoperate. The .NET programs run inside the .NET execution runtime rather than on a particular hardware or operating system platform. .NET is also the collective name given to various software components that are built on the .NET platform. The components that make up the .NET environment are referred to as the .NET Framework.

2.1.1 Architecture

The .NET Framework has two main components: the common language runtime and the unified .NET Framework class library.

The common language runtime (CLR) is the foundation of the .NET Framework. You can think of the runtime as an agent that manages code at execution time, providing core services such as memory management, thread management, and remoting, while also enforcing strict type safety and security. In fact, the concept of code management is one of the cornerstones of .NET architecture. Code that targets the runtime is known as *managed code*, and code that does not target the runtime is known as *unmanaged code*.

The *class library*, the other main component of the .NET Framework, is a comprehensive, object-oriented collection of reusable types that you can use to develop applications ranging from traditional command-line or graphical user interface (GUI) applications to applications based on the latest innovations provided by ASP.NET, such as Web Forms and XML Web services.

Typically, .NET applications use ADO.NET classes to access and manipulate database objects.

The common language runtime manages memory, thread execution, code execution, code safety verification, compilation, and other system services. These features are intrinsic to the managed code that runs on the common language runtime.

.NET programs are not compiled directly into executable code but are compiled into an intermediary language known as Microsoft Intermediate Language (MSIL or IL). Later, at program execution, the CLR loads the code into the runtime environment and a just-in-time compiler (JIT) compiles the IL into native executable code. This native code is then executed by the runtime's execution engine.

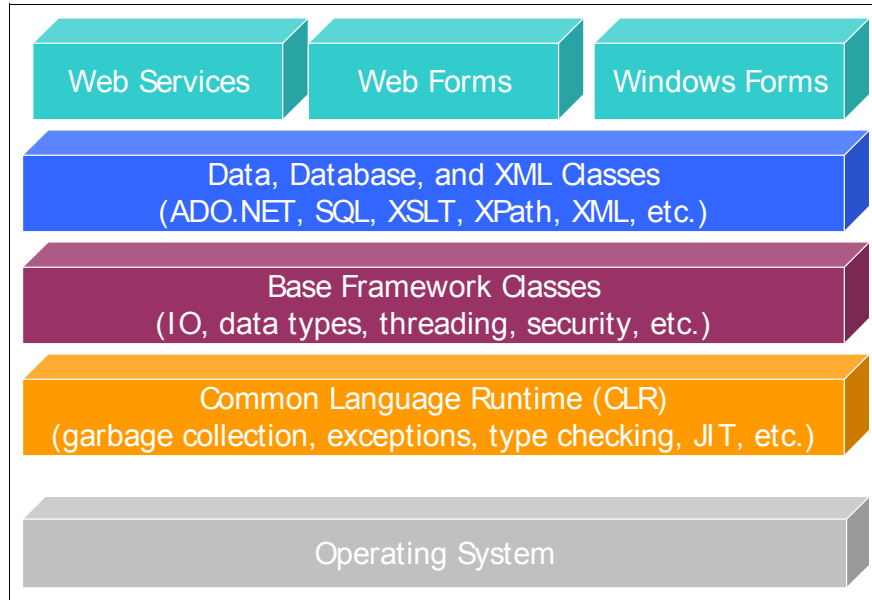


Figure 2-1 .NET Framework components

2.1.2 Platform support

The Microsoft .NET Framework is designed for Windows operating systems. To run any .NET application, the client or server must have a runtime called the *.NET redistributable*, which is freely available from the Microsoft Web site. For configuring server-side applications, the recommendation is to use Windows 2000 Server or higher. The following list describes the Windows platforms that are supported by the .NET framework and the two .NET managed providers we discuss in this book.

Table 2-1 .NET and DB2 providers operating systems supported

Operating system	.NET Framework	IBM DB2 UDB for iSeries .NET provider	IBM DB2 for LUW .NET provider
Microsoft Windows 98 and Editions	YES	NO	YES
Microsoft Windows Millennium Edition	YES	NO	YES
Microsoft Windows NT® 4.0 Workstation with Service Pack 6.0a or later	YES	YES	YES
Microsoft Windows NT 4.0 Server with Service Pack 6.0a or later	YES	YES	YES
Microsoft Windows 2000 Professional	YES	YES	YES
Microsoft Windows 2000 Server family	YES	YES	YES
Microsoft Windows XP Home Edition	YES	NO	YES
Microsoft Windows XP Professional	YES	YES	YES
Microsoft Windows Server 2003 family	YES	YES	YES

2.2 Common Language Runtime (CLR)

Microsoft .NET supports multiple languages on the Microsoft Windows platform. The .NET languages follow the *Common Language Specification (CLS)*, the minimum set of features that compilers must support to target the .NET runtime. Currently, .NET supports more than 20 languages, including vendor-supported languages.

Visual Studio .NET comes with following languages developed and supported by Microsoft:

- ▶ Visual Basic .NET
- ▶ Microsoft C#
- ▶ Microsoft J#
- ▶ Visual C++

Moreover, Visual Studio .NET supports scripting languages such as JScript.NET and VBScript. Numerous other third-party languages also are available.

The .NET Framework supports so many languages because each language compiler translates the source code into MSIL, which is a CPU-independent set of instructions that can be efficiently converted to native code.

2.2.1 Runtime execution environment

The CLR is the heart of the .NET Framework and provides a runtime environment for .NET applications. The CLR provides a fundamental set of services that all programs can use. It can compile managed code once, then run on any CPU and operating system that supports the runtime. The CLR runs intermediate language, which is created from any .NET programming language, such as VB.NET and C#. Figure 2-2 shows the role of the CLR when executing managed code.

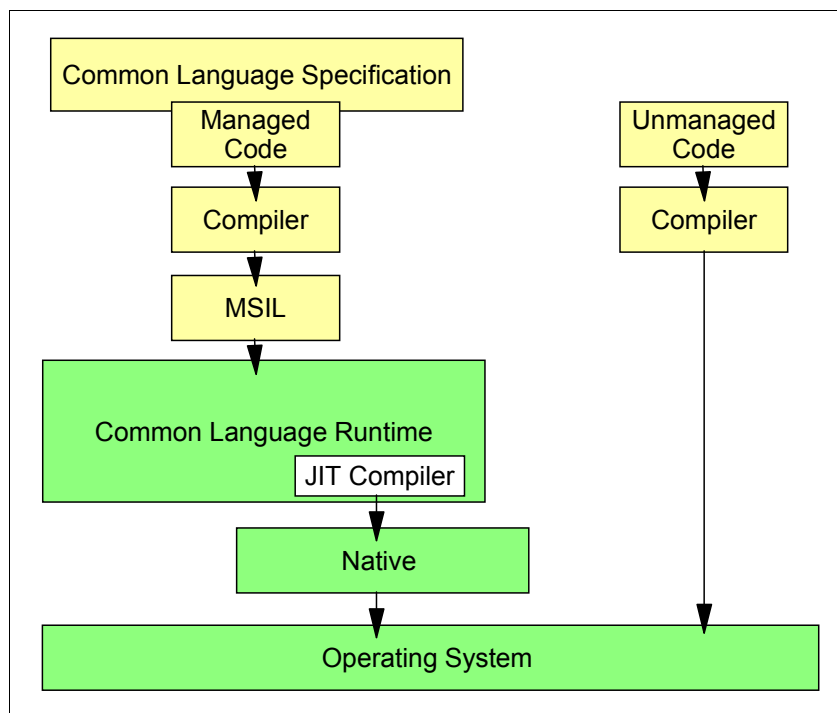


Figure 2-2 Common Language Runtime in .NET Framework

In the .NET Framework, managed code is the code that follows the CLS. The managed code gets compiled into the Microsoft Intermediate Language (MSIL). The CLR provides a JIT compiler that compiles the MSIL into the machine language and then runs it, because all programs use the common services in the CLR, no matter which language they are written in. The CLR follows the *Common Type System (CTS)*, which is a master set of data types. Because of the CTS, managed code written in one language can interoperate with programs written in another CLR language.

Some of the features of Common Language Runtime are:

- ▶ Garbage collection
- ▶ Cross-language integration
- ▶ Base class library support
- ▶ Thread support
- ▶ Exception manager
- ▶ Security
- ▶ IL to native
- ▶ Class loader

2.3 Class libraries

The .NET Framework class library is a set of classes, interfaces, and value types providing a foundation to develop controls, components, and applications. These classes are organized into namespaces, and each namespace contains classes related to specific functionality. Table 2-2 shows some of these namespaces and the functionality covered by their classes.

Table 2-2 Some of most widely used namespaces in .NET applications

Namespace	Description
System	Contains classes that define commonly used value and reference data types, events and event handlers, interfaces, attributes, and exceptions, and other classes that provide services supporting data type conversion, method parameter manipulation, mathematics, remote and local program invocation, application environment management, and supervision of managed and unmanaged applications.
System.Collections	Contains interfaces and classes that define various collections of objects, such as lists, queues, arrays, hash tables, and dictionaries.
System.Data	Consists mostly of the classes that constitute the ADO.NET architecture. We discuss ADO.NET in detail in Chapter 3, “ADO .NET object hierarchy” on page 17.
System.Drawing	Provides access to GDI+ basic graphics functionality.
System.Globalization	Contains classes that define culture-related information, including the language, the country or region, calendars in use, format patterns for dates, currency, and numbers, and the sort order for strings.
System.IO	Contains types that enable synchronous and asynchronous reading and writing of data streams and files.
System.Net	Provides a simple programming interface for many of the protocols used with networks.
System.Reflection	Contains classes and interfaces that provide a managed view of loaded types, methods, and fields, with the ability to dynamically create and invoke types.

Namespace	Description
System.Resources	Provides classes and interfaces that enable developers to create, store, and manage various culture-specific resources used in an application.
System.Security	Provides the underlying structure of the .NET Framework security system, including base classes for permissions.
System.Threading	Provides classes and interfaces that enable multi-threaded programming.
System.Web	Supplies classes and interfaces that enable browser-server communication.
System.Web.Services	Consists of classes that enable creation of XML Web services using ASP.NET and XML Web service clients.
System.Windows.Forms	Contains classes for creating Windows-based applications that take full advantage of the rich user interface features that are available in the Microsoft Windows operating system.
System.Xml	Provides standards-based support for processing XML.



ADO .NET object hierarchy

This chapter discusses the following topics:

- ▶ An overview of ADO.NET, explaining the principal components of the ADO.NET architecture. The examples in this chapter use the IBM DB2 UDB for iSeries .NET provider that is discussed in Chapter 4, “IBM DB2 UDB for iSeries .NET provider” on page 33.
- ▶ Using ADO.NET in connected mode.
- ▶ Using ADO.NET in disconnected mode.

3.1 Overview of ADO .NET

ADO.NET is a suite of data-access technologies that are included in the .NET Framework class libraries. ADO.NET helps applications connect to a database and has been designed to be the single data access model used by all server processes and applications running on the Microsoft platform. Figure 3-1 shows the relationship between various applications based on the .NET framework, ADO.NET, and a database.

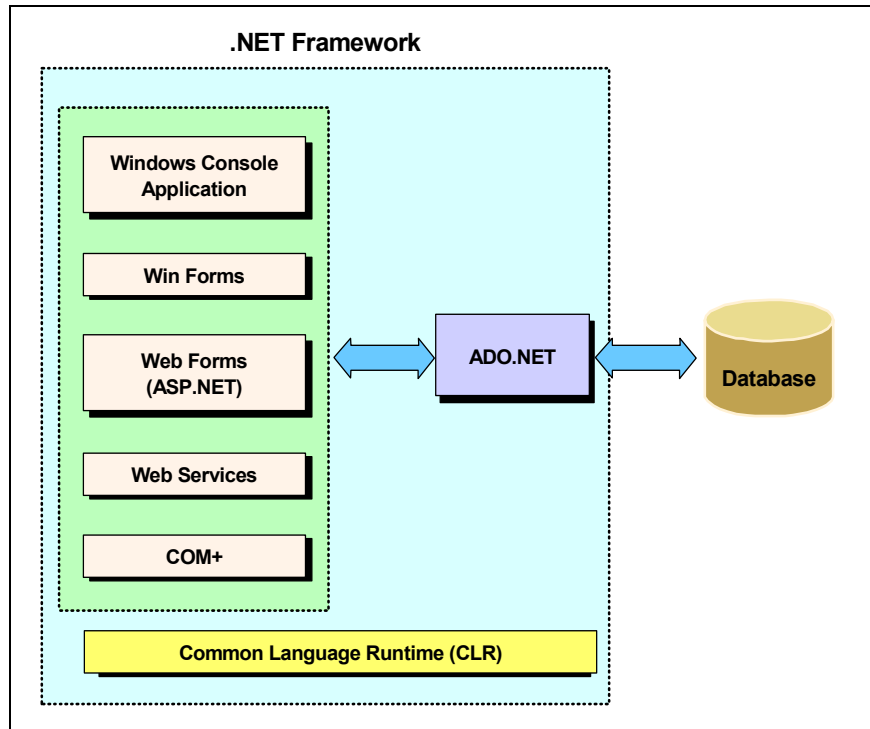


Figure 3-1 ADO.NET

ADO.NET replaces the ADO Recordset object with a suite of objects (*DataTable*, *DataSet*, *DataAdapter*, and *DataReader*) that interact to access relational database systems. A *DataTable* is similar to the ADO Recordset representing a collection of rows from a single table. A *DataSet* is a collection of *DataTable* objects with relationships and constraints that bind these tables together. In disconnected mode, a *DataSet* is an in-memory relational structure supporting database operations. This structure is natively disconnected; a *DataSet* does not have to know the underlying data source that might have been used to populate it.

The *DataAdapter* is the object that enables *DataSet*-datasource communication, channeling the data. It has batch update features that are ideal for a disconnected scenario. Figure 3-2 on page 19 shows the various components of ADO.NET.

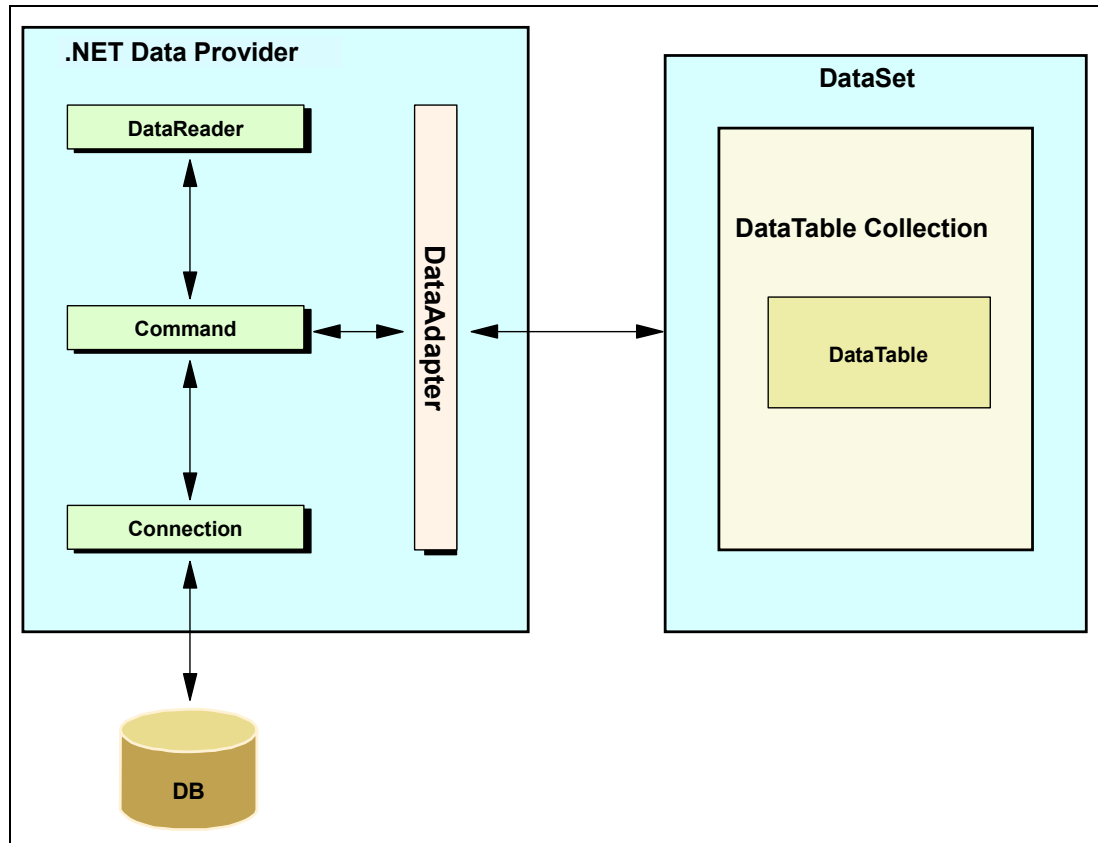


Figure 3-2 ADO.NET architecture

3.1.1 Connection

The Connection object in ADO.NET is used to connect to a database and can be used to control the transactions. The Connection objects are different for different providers but they serve one purpose. For example, the Connection object for the DB2 UDB for iSeries .NET provider is called *iDB2Connection*; the Connection object for the DB2 for LUW .NET provider is called *DB2Connection*; and the System.Data.OleDb provider's Connection object is called *OleDbConnection*.

Connections can be opened in one of two ways:

- Explicitly by calling the Open method on the connection
- Implicitly when using a DataAdapter

Table 3-1 and Table 3-2 on page 20 describe some important interfaces of the Connection object.

Table 3-1 Connection object properties

Public property	Description
ConnectionString	This is required for making a connection with a database. It requires the database source name and other parameters. For example, with the DB2 UDB for iSeries .NET provider you can specify a ConnectionString property with Connection cnn as: <pre>cnn.ConnectionString = "DataSource=myiSeries;"</pre>

Table 3-2 Connection object methods

Public method	Description
Open	Opens a database connection to the data source that is specified in a <code>ConnectionString</code> property, for example: <code>cnn.Open();</code> The <code>Connection</code> object throws an exception if it fails to open a database connection.
Close	Used to close the database connection. For example: <code>cnn.Close();</code> You should always close your connection when you are finished using it.
CreateCommand	Returns a <code>Command</code> object associated with the connection, which can be used to perform SQL operations on a database. For example: <code>IDB2Command cmd = cnn.CreateCommand();</code>
BeginTransaction	Begins a transaction at the local level.

3.1.2 Command

The `Command` object is used to execute SQL statements or Stored Procedures on a database (data source). Example 3-1 shows how to create a command object and assign its `Connection` property.

Example 3-1 Command object

```
//Create a command object
IDB2Command cmd = new IDB2Command();

//Assign the connection for where to perform the operation
cmd.Connection =cnn;

//Assign the CommandText
cmd.CommandText = "SELECT * FROM ORG";

//Open the command
cnn.Open();

//Read the data by executing the CommandText against the data source
IDB2DataReader reader = cmd.ExecuteReader();
```

Table 3-3 and Table 3-4 on page 21 describe some important interfaces of the `Command` object.

Table 3-3 Command object properties

Public property	Description
CommandType	Describes whether the <code>Command</code> object will execute an SQL statement or Stored Procedure, or select from a set of tables.
CommandText	Describes the SQL statement or Stored Procedure to execute against a database. The default value of the <code>CommandType</code> property is <code>CommandType.Text</code> . For example: <code>cmd.CommandText = "select * from STAFF";</code> <code>cmd.CommandType = CommandType.Text;</code>

Table 3-4 Command object methods

Public method	Description
CreateParameter	Used for handling parameters. The parameter could be input-only, output-only, bidirectional, or a stored procedure return value parameter.
ExecuteNonQuery	Can be used to perform INSERT, UPDATE, or DELETE SQL operations on a database. This method returns the number of rows that are affected after executing the SQL statement. For example: <pre>cmd.Connection.Open(); cmd.ExecuteNonQuery();</pre>
ExecuteReader	Used for reading results by executing a SELECT statement on a database or calling a stored procedure that returns result data.
ExecuteScalar	Used for retrieving a single value from a database. This reduces overhead required for the ExecuteReader method when the result contains only a single value. For example: <pre>cmd.CommandText = "select count(*) from STAFF"; Int32 count = (Int32) cmd.ExecuteScalar();</pre>

3.1.3 DataReader

The DataReader class is used for reading data into your client application by reading a forward-only stream of rows from a database. The code in Example 3-2 shows how to create an iDB2DataReader object and populate it to display the result on the Windows console.

Example 3-2 Create iDB2DataReader object

```
iDB2DataReader reader;
reader = cmd.ExecuteReader();
// Populate result
while (reader.Read())
{
    Console.WriteLine(reader.GetString(1) + ", " + reader.GetString(2));
}
// close reader
reader.Close();
```

Table 3-5 and Table 3-6 describe important properties and methods of the DataReader class.

Table 3-5 DataReader class properties

Public property	Description
FieldCount	Returns the number of columns in the current row.
HasRows	Indicates whether DataReader has one or more rows.

Table 3-6 DataReader methods

Public method	Description
Read	Used to read records one by one. This method automatically advances the cursor to the next record and returns true or false, indicating whether the DataReader read any rows.
Close	Closes the DataReader. Always close your DataReader when you are through.
Getxxxx	Used to get data of type xxxx. For example, the <i>GetBoolean</i> method is used to get Boolean records, and the <i>GetChar</i> method is used to get <i>char</i> -type data.

3.1.4 DataAdapter

A DataAdapter is used between a DataSet and a database such as DB2 UDB for iSeries. The DataAdapter performs SELECT, INSERT, UPDATE, and DELETE operations for loading or unloading the data. Example 3-3 shows how to create an iDB2DataAdapter.

Example 3-3 Create an iDB2DataAdapter

```
iDB2Connection cnn =  
    new iDB2Connection("DataSource=myiSeries;DefaultCollection=SAMPLEDB");  
DataSet ds = new DataSet();  
iDB2DataAdapter adpt = new iDB2DataAdapter();  
adpt.SelectCommand = new iDB2Command("select * from staff", cnn);  
adpt.Fill(ds);  
//--- Code to perform further Operations on a dataset---
```

Table 3-7 and Table 3-8 show some important DataAdapter public properties and methods.

Table 3-7 DataAdapter properties

Public property	Description
DeleteCommand	Deletes records from a database using a SQL statement or a Stored Procedure. For example: iDB2DataAdapter adpt = new iDB2DataAdapter (); iDB2Command cmd; cmd = new iDB2Command("DELETE FROM Customers WHERE CustomerID = ' '", cnn); adpt.DeleteCommand = cmd;
InsertCommand	Inserts new records into a database using a SQL statement or Stored Procedure.
SelectCommand	Selects records from a database using a SQL statement or Stored Procedure.
UpdateCommand	Updates records in a database using a SQL statement or Stored Procedure.

Table 3-8 DataAdapter methods

Public method	Description
Fill	Used to fill records in a DataSet. For example: adpt.Fill(dataset); //fills the dataset
Update	Used to update rows in the DataSet and a database by performing INSERT, DELETE, or UPDATE operations.

3.1.5 DataSet

The DataSet class represents a in-memory cache of data retrieved from a database. The DataSet is used to improve the overall performance of the application, because it minimizes server trips to a database. Example 3-4 shows sample code using a DataSet.

Example 3-4 DataSet class

```
iDB2Connection cnn =  
    new iDB2Connection("DataSource=myiSeries;DefaultCollection=SAMPLEDB;");  
DataSet ds = new DataSet();  
iDB2DataAdapter adpt = new iDB2DataAdapter();  
adpt.SelectCommand = new iDB2Command("select * from staff", cnn);  
adpt.Fill(ds);  
//---Code to perform further Operations on dataset---
```

Table 3-9 and Table 3-10 show some important DataSet public properties and methods.

Table 3-9 DataSet properties

Public property	Description
DataSetName	Gets or sets the DataSet name.

Table 3-10 DataSet methods

Public method	Description
AcceptChanges	Used to commit changes made to the DataSet.
Clear	Clears the contents of the DataSet.
GetXML	Gets an XML representation of data in the DataSet.
ReadXml	Reads XML schema and XML into the DataSet.
WriteXml	Writes XML schema and XML into the DataSet.

3.2 Connected mode

In connected mode, we use SQL queries or stored procedures to directly select, insert, update, and delete data objects in the iSeries DB2 UDB database.

The advantages of using stored procedures include:

- ▶ Reduced network usage between client and server
- ▶ Enhanced hardware and software capabilities
- ▶ Improved security
- ▶ Reduced development cost and increased reliability
- ▶ Centralized security, administration, and maintenance for common routines

Using the IBM DB2 UDB for iSeries .NET provider, you can call stored procedures, which take input parameters and return results in output parameters. Stored procedures can also return one or more result sets. A stored procedure call is demonstrated using the following steps:

1. Before starting actual coding, create the stored procedure in Example 3-5 on your iSeries server. Our EMP_INFO procedure takes an employee ID as input, selects the employee's record from the STAFF table, and returns the employee's salary, name, and job.

Example 3-5 Create stored procedure EMP_INFO

```
CREATE PROCEDURE SAMPLEDB.EMP_INFO (IN idno INT,  
                                     OUT empsalary decimal(7, 2),  
                                     OUT empname VARCHAR(9),  
                                     OUT empjob CHAR(5)) LANGUAGE SQL  
  
BEGIN  
    DECLARE year INT DEFAULT 0;  
    DECLARE c1 CURSOR FOR  
    SELECT salary, years, job, name FROM sampledb.staff WHERE ID =  
        idno;  
    OPEN c1;  
    FETCH c1 INTO empsalary, year, empjob, empname;  
    CLOSE c1;  
    RETURN year;  
END
```

In .NET code, we assume that a connection is already created. To create a command object whose CommandType property is set to CommandType.StoredProcedure and CommandText to the name of the stored procedure, use the code shown in Example 3-6.

Example 3-6 Create command object

```
iDB2Command cmd = cnn.CreateCommand();
cmd.CommandText = "SAMPLEDB.EMP_INFO";
cmd.CommandType = CommandType.StoredProcedure;
```

-
2. The input and output parameters can be assigned using the iDB2Parameter object. Here you define details of each parameter including direction and type, as shown in Example 3-7.

Example 3-7 Create in and out parameters

```
// create in and out parameters
iDB2Parameter parm = cmd.Parameters.Add("@empyears", iDB2DbType.iDB2Integer);
cmd.Parameters["@empyears"].Direction = ParameterDirection.ReturnValue;

parm = cmd.Parameters.Add("@empid", iDB2DbType.iDB2Integer);
cmd.Parameters["@empid"].Direction = ParameterDirection.Input;
cmd.Parameters["@empid"].Value = empId;

parm = cmd.Parameters.Add("@empsalary", iDB2DbType.iDB2Decimal);
cmd.Parameters["@empsalary"].Precision = 7;
cmd.Parameters["@empsalary"].Scale = 2;
cmd.Parameters["@empsalary"].Direction = ParameterDirection.Output;

parm = cmd.Parameters.Add("@empname", iDB2DbType.iDB2VarChar,9);
cmd.Parameters["@empname"].Direction = ParameterDirection.Output;

parm = cmd.Parameters.Add("@empjob", iDB2DbType.iDB2Char,5);
cmd.Parameters["@empjob"].Direction = ParameterDirection.Output;
```

-
-
3. Execute the stored procedure using the ExecuteNonQuery method of the command object as shown in Example 3-8.

Example 3-8 Call the stored procedure

```
// Call the stored procedure
cmd.ExecuteNonQuery();
```

-
-
-
4. Finally, retrieve output parameters using the command object and display the output as shown in Example 3-9.

Example 3-9 Retrieve output parameter using the command object

```
// Retrieve output parameters
Decimal salary = (Decimal)cmd.Parameters["@empsalary"].Value;
Int32 years = (Int32)cmd.Parameters["@empyears"].Value;
String name = (String)cmd.Parameters["@empname"].Value;
String job = (String)cmd.Parameters["@empjob"].Value;

// Display details of the employee
Console.WriteLine("    Employee Name : " + name);
Console.WriteLine("    Employee Job : " + job);
Console.WriteLine("    Employee Salary : " + String.Format("{0:f2}",salary));
Console.WriteLine("    Years Served : " + years);
```

3.3 Disconnected mode

In disconnected mode, we use ADO.NET methods to add, edit, and remove data objects in a cached DataRow, DataTable, or even an entire DataSet, then merge the changes back into the data source. This technique is very useful when developing applications for smart devices such as smart phones, PDAs, and other appliances.

We begin by writing a Winform application that shows a disconnected scenario:

1. Open Microsoft Visual Studio .NET and click **New Project**. Then select **Visual C# Projects** and **Windows Application Template**. Fill out the Name text box and select a location for your project. Click **OK**.
2. Open Form1.cs in design mode and drag a DataGrid and four buttons from the Toolbox. Your design should be similar to the picture shown in Figure 3-3.

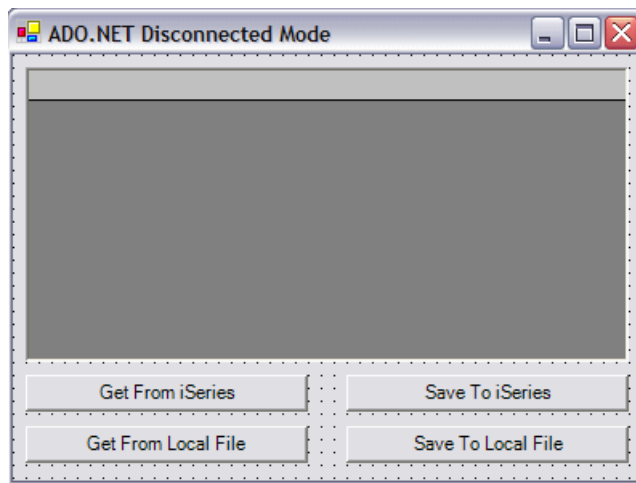


Figure 3-3 ADO.NET Disconnected Mode Windows form sample

3. Name the controls using the names shown in Table 3-11.

Table 3-11 DataSet methods

Control Name	Description
dgDept	The DataGrid to show data.
btnGetFromDB	Button with text "Get From iSeries"
btnSaveToDB	Button with text "Save To iSeries"
btnGetFromXml	Button with text "Get From Local File"
btnSaveToXml	Button with text "Save To Local File"

4. From the Toolbox, add an **OpenFileDialog** control and a **SaveFileDialog** control by dragging these to your window design area.
5. Add a project reference to **IBM.Data.DB2.iSeries**.
6. Open the Form1.cs code view and add the following using directives as shown in Example 3-10 on page 26.

Example 3-10 Using directives needed for ADO.NET Disconnected Mode sample

```
using System.Data;
using System.IO;
using System.Text;
using System.Xml;
using System.Diagnostics;
using IBM.Data.DB2.iSeries;
```

7. Add the following variables to Form1.cs as shown in Example 3-11:

Example 3-11 Attributes to add for ADO.NET Disconnected Mode sample

```
private iDB2DataAdapter da;
private iDB2CommandBuilder builder;
private DataSet dsDept = new DataSet();
private string connectionString = "DataSource=myiSeries;DefaultCollection=SAMPLEDB;";
private string tableName = "Department";
```

8. Build a method to log possible exceptions with our iSeries operations. Use the event log provided by the Windows OS. The LogDB2Exception method is shown in Example 3-12.

Example 3-12 Log exceptions method for ADO.NET Disconnected Mode sample

```
private void LogDB2Exception(iDB2Exception ex)
{
    EventLog el = new EventLog();
    el.Source = "AdoDisconnectedApp";
    string strMessage;
    strMessage = "Exception Number : " + ex.MessageCode.ToString()
        + "(" + ex.MessageDetails + ") has occurred";
    el.WriteEntry(strMessage, EventLogEntryType.Error);
    foreach(iDB2Error er in ex.Errors)
    {
        strMessage = "Message : "+ er.Message
            + " Code : "+ er.MessageCode.ToString()
            + " State : "+ er.SqlState;
        el.WriteEntry(strMessage, EventLogEntryType.Error);
    }
}
```

9. Write your method to read data from the iSeries database. Use the LogDB2Exception method in case of a problem connecting to or reading from the database (Example 3-13).

Example 3-13 Reading data for ADO.NET Disconnected Mode sample

```
private DataSet RetrieveDataSetFromDB()
{
    DataSet ds = new DataSet();
    try
    {
        using (iDB2Connection cnn =
            new iDB2Connection(this.connectionString) )
        {
            iDB2Command cmd = new iDB2Command();
            cmd.Connection = cnn;
            cmd.CommandText = "Select DeptNo, DeptName, admrdept from Department order by DeptNo";
            cnn.Open();
            //create Data adapter
            this.da = new iDB2DataAdapter(cmd.CommandText,cnn);
            this.builder = new iDB2CommandBuilder(this.da);
```

```

        //fill DataSet with data
        this.da.Fill(ds, this.tableName);
        cnn.Close ();
    }
}
catch (iDB2Exception iex)
{
    this.LogDB2Exception(iex);
}
catch (Exception ex)
{
    MessageBox.Show(ex.Message);
}
return ds;
}

```

10. Link this method with the button btnGetFromDB click event. Switch to design mode and click **Get From iSeries**. Copy the code from Example 3-14 into your code.

Example 3-14 btnGetFromDB click event code for ADO.NET Disconnected Mode sample

```

private void btnGetFromDB_Click(object sender, System.EventArgs e)
{
    this.dsDept = this.RetrieveDataSetFromDB();
    this.dgDept.DataSource = this.dsDept;
    this.dgDept.DataMember = this.dsDept.Tables[0].TableName;
    this.dgDept.CaptionText = this.dsDept.Tables[0].TableName + " from iSeries";
}

```

11. You can now generate and run your application, but first, we show how to complete the functionality. First, add a method to save a DataSet as an XML file in the local file system. Copy the PersistDataSetToXml method shown in Example 3-15 into your code.

Example 3-15 Code to save DataSet as XML file for ADO.NET Disconnected Mode sample

```

private void PersistDataSetToXml(DataSet ds, string path)
{
    if(ds == null)
        return;
    FileStream fs = new FileStream(path, FileMode.Create);
    XmlTextWriter writer = new XmlTextWriter(fs, Encoding.Unicode);
    ds.WriteXmlSchema(Path.ChangeExtension(path, ".xsd"));
    ds.WriteXml(writer, XmlWriteMode.DiffGram);
    writer.Close();
}

```

12. Copy the RetrieveDataSetFromXml method shown in Example 3-16 into your code. This method reads an XML file from the local file system into a DataSet.

Example 3-16 Code to Load a DataSet from a XML file for ADO.NET Disconnected Mode sample

```

private DataSet RetrieveDataSetFromXml(DataSet ds, string path)
{
    FileStream fs = new FileStream(path, FileMode.Open);
    XmlTextReader reader = new XmlTextReader(fs);
    ds.ReadXmlSchema(Path.ChangeExtension(path, ".xsd"));
    ds.ReadXml(reader, XmlReadMode.DiffGram);
    reader.Close();
    return ds;
}

```

13. Copy the PersistDataSetToDB method shown in Example 3-17 into your code. This method calls Update to update data on the iSeries database.

Example 3-17 Code to Update DataSet on iSeries for ADO.NET Disconnected Mode sample

```
private void PersistDataSetToDB(DataSet ds)
{
    try
    {
        using (iDB2Connection cnn =
            new iDB2Connection(this.connectionString) )
        {
            iDB2Command cmd = new iDB2Command();
            cmd.Connection = cnn;
            cmd.CommandText = "Select DeptNo, DeptName, admrdept from Department order by DeptNo";
            cnn.Open();

            //create Data adapter
            this.da = new iDB2DataAdapter(cmd.CommandText,cnn);
            this.builder = new iDB2CommandBuilder(this.da);

            //update database
            int rowsAffected = this.da.Update(ds, this.tableName);
            MessageBox.Show(rowsAffected.ToString() + " rows affected by update.");
            cnn.Close();
        }
    }
    catch (iDB2Exception iex)
    {
        this.LogDB2Exception(iex);
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message.ToString());
    }
    this.dsDept = this.RetrieveDataSetFromDB();
}
```

14. Introduce the code for the click events of the other buttons. Copy the methods shown in Example 3-18 into your code.

Example 3-18 Code for click events of buttons on Form1.cs for ADO.NET Disconnected Mode sample

```
private void btnSaveToXml_Click(object sender, System.EventArgs e)
{
    if (this.saveFileDialog1.ShowDialog() == DialogResult.OK)
        this.PersistDataSetToXml(this.dsDept, this.saveFileDialog1.FileName);
    this.dgDept.CaptionText = this.dsDept.Tables[0].TableName + " saved to local XML
file";
}

private void btnGetFromXml_Click(object sender, System.EventArgs e)
{
    if (this.openFileDialog1.ShowDialog() == DialogResult.OK)
    {
        this.RetrieveDataSetFromXml(this.dsDept, this.openFileDialog1.FileName);
        this.dgDept.DataSource = this.dsDept;
        this.dgDept.DataMember = this.dsDept.Tables[0].TableName;
        this.dgDept.CaptionText = this.dsDept.Tables[0].TableName + " from local XML file";
    }
}
```

```
private void btnSaveToDB_Click(object sender, System.EventArgs e)
{
    this.PersistDataSetToDB(this.dsDept);
    this.dgDept.CaptionText = this.dsDept.Tables[0].TableName + " saved to iSeries";
}
}
```

15. Ensure that your changes are not lost by adding the code shown in Example 3-19.

Example 3-19 Code for Closing event of Form1.cs for ADO.NET Disconnected Mode sample

```
private void Form1_Closing(object sender, System.ComponentModel.CancelEventArgs e)
{
    if(this.dsDept.HasChanges())
    {
        if(DialogResult.No == MessageBox.Show("You have changes not saved to the Database!
        If you haven't saved to a local file, exiting now will result in your changes being lost.
        Exit anyway?", "Please Confirm", MessageBoxButtons.YesNo, MessageBoxIcon.Question))
            e.Cancel = true;
    }
}
}
```

16. We can now generate and run this project. Press the F5 key in Visual Studio. When the program is running, click **Get From iSeries**. You may be prompted to log on to the iSeries (if you do not include your credentials in connection string). A window appears that looks similar to Figure 3-4.

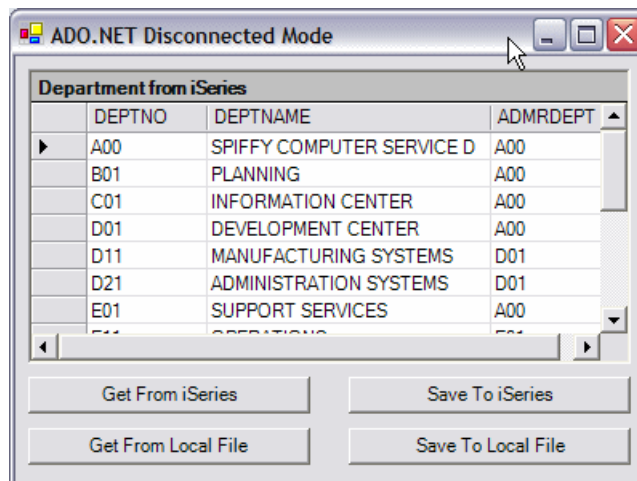


Figure 3-4 ADO.NET Disconnected Mode Windows form with data from iSeries loaded

17. You can make changes on the DataGrid. For example, change the name of department E01 from SUPPORT SERVICES to SUPPORT SERVICES CHANGED and click **Save To Local File**. You are prompted for a location and a name for the local file. Type department.xml as the file name and select a folder to save this file into, as shown in Figure 3-5 on page 30.

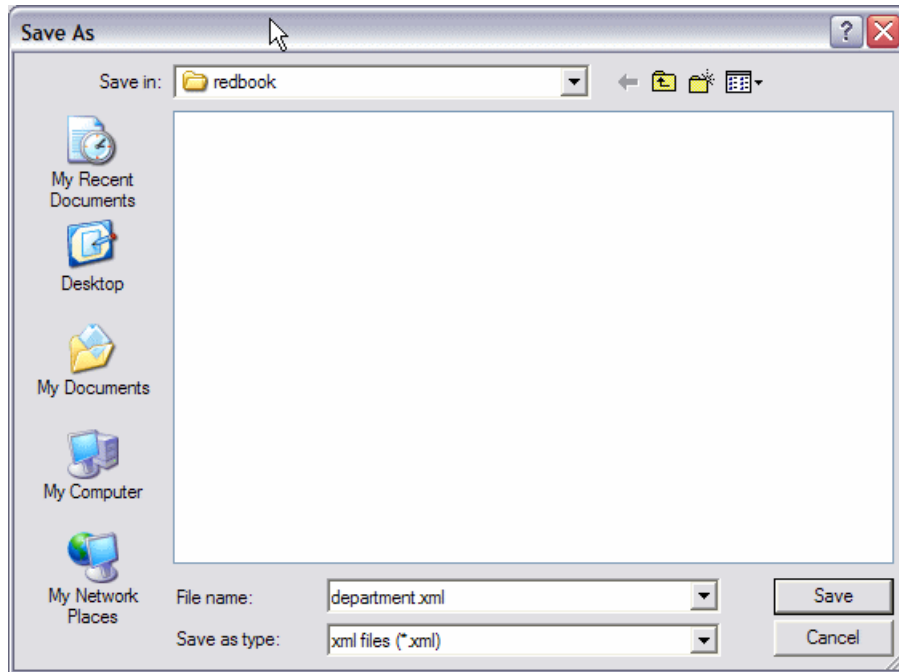


Figure 3-5 ADO.NET Disconnected Mode Save As window

18. Click **Save** to save the DataSet onto your local file system in DiffGram format. Another file with .xsd extension is saved, too. See the formats in Figure 3-6 and Figure 3-7.

```
- <Department diffgr:id="Department7" msdata:rowOrder="6" diffgr:hasChanges="modified">
  <DEPTNO>E01</DEPTNO>
  <DEPTNAME>SUPPORT SERVICES CHANGED</DEPTNAME>
  <ADMRDEPT>A00</ADMRDEPT>
</Department>
```

Figure 3-6 XML data fragment: note the DiffGram entries with the modified stamp

```
- <diffgr:before>
- <Department diffgr:id="Department7" msdata:rowOrder="6">
  <DEPTNO>E01</DEPTNO>
  <DEPTNAME>SUPPORT SERVICES</DEPTNAME>
  <ADMRDEPT>A00</ADMRDEPT>
</Department>
</diffgr:before>
```

Figure 3-7 XML data fragment: note the DiffGram entries with prior data

19. Close the window and affirm to save changes.

20. Press F5, and when the application is running, click **Get From Local File**. The DataSet will be restored from XML. You can see the changes made are there.

21. Click **Save To iSeries** to open a dialog informing you how many rows have been affected by this update.



Part 2

Providers

In this part we introduce and explain the .NET providers that are available to access DB2 UDB for iSeries:

- ▶ In Chapter 4, “IBM DB2 UDB for iSeries .NET provider” on page 33, we concentrate on the native IBM DB2 UDB for iSeries .NET provider.
- ▶ In Chapter 5, “IBM DB2 for LUW .NET provider” on page 177, we discuss the IBM DB2 for LUW (Linux, UNIX, and Windows) .NET provider.
- ▶ In Chapter 6, “Selecting the .NET provider” on page 225, we evaluate both providers.



IBM DB2 UDB for iSeries .NET provider

In this chapter we discuss the IBM DB2 UDB for iSeries .NET provider, which was first included with the iSeries Access for Windows product (licensed program 5722-XE1) in its V5R3M0 release.

We include information to help you get started, some coding examples, and a discussion of more advanced topics. Because we understand performance is an important feature of any application, we have some tips that will help you tune your application. Finally, we discuss some differences between ADO.NET and OLE DB, and offer suggestions on how to migrate from OLE DB to ADO.NET.

Note: The IBM DB2 UDB for iSeries .NET provider is also referred to by the namespace it defines, *IBM.Data.DB2.iSeries*. The two terms are used interchangeably.

4.1 Introduction

The IBM DB2 UDB for iSeries .NET provider is part of the iSeries Access for Windows product, starting in its V5R3M0 release. It uses the optimized host database server job (QZDASOINIT) to perform database requests on the iSeries. Because it was written especially for the iSeries and uses the optimized server, it can take advantage of improvements that are made especially for the iSeries. The provider is a component of iSeries Access for Windows that does not require the iSeries Access Family (5722-XW1) license.

The namespace for the provider is IBM.Data.DB2.iSeries, and it includes a full set of ADO.NET classes that enable your .NET application to access an iSeries database using commands, queries, and stored procedure calls. You can read more about ADO.NET at the MSDN Library Web site:

<http://msdn.microsoft.com/library/>

When you reach this Web page, select **.NET Development** → **Data Access and Storage** → **ADO.NET**.

The Microsoft Web site has a wealth of information about ADO.NET, including examples using the Microsoft ADO.NET providers. In many cases, you can use the coding examples provided on the site and substitute the iDB2- prefix for the Sql- or OleDb- prefix on the class names. For instance, where their example references a SqlConnection or OleDbConnection, you can change the example to instead reference iDB2Connection.

The IBM DB2 UDB for iSeries provider (IBM.Data.DB2.iSeries) has been tested with the Microsoft .NET Framework 1.0 and 1.1.

4.2 IBM.Data.DB2.iSeries architecture

In this section, we discuss how the IBM DB2 UDB for iSeries provider fits into the ADO.NET picture. We explain how the provider communicates with the iSeries server, and list the features supported by the provider.

4.2.1 ADO.NET interfaces

The IBM.Data.DB2.iSeries provider supports all required interfaces of an ADO.NET provider. Figure 4-1 on page 35 shows how the IBM.Data.DB2.iSeries provider classes fit into the ADO.NET object model.

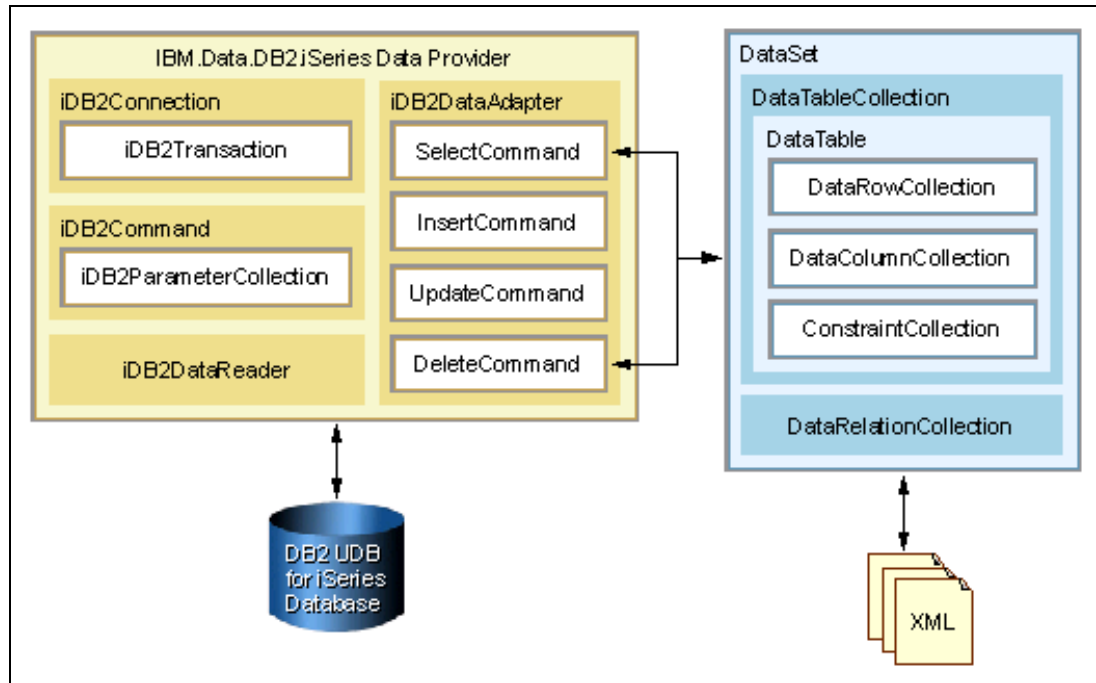


Figure 4-1 IBM.Data.DB2.iSeries provider object model

4.2.2 Host server jobs

When you run applications that use the IBM.Data.DB2.iSeries .NET provider, much of the work is performed by the iSeries server on behalf of your application. This is accomplished with the help of host server jobs that run on the iSeries. The provider handles transferring the commands and data back and forth between your PC and the host server jobs. Because it is part of iSeries Access for Windows, the provider can take advantage of its security and communication features.

The provider communicates with the iSeries by using the optimized host database server. The host server has a prestart job, QZDASOINIT, which normally runs under the QUSRWRK subsystem. When using secure connections (see “SSL” on page 50), the host server job QZDASSINIT is used instead. Other host server jobs are also used when making a connection. The host servers are installed as part of the operating system 5722SS1 option 12 (Host Servers).

Before you can use the IBM.Data.DB2.iSeries .NET provider to communicate with your iSeries host server jobs, the server jobs must be active.

- To start the host servers, use the STRHOSTSVR CL command from your 5250 workstation or emulator, or from your PC desktop select **Start** → **Programs** → **IBM iSeries Access for Windows** → **iSeries Navigator**. In the iSeries Navigator window, select **My Connections** → *myiSeries* → **Network** → **Servers** → **iSeriesAccess**. The server names and their status are listed in the right panel. To start or stop a server, right-click the server and select **Start** or **Stop**. The iSeries Access server jobs are shown in Figure 4-2 on page 36.

Note: Substitute your iSeries server name for *myiSeries*.

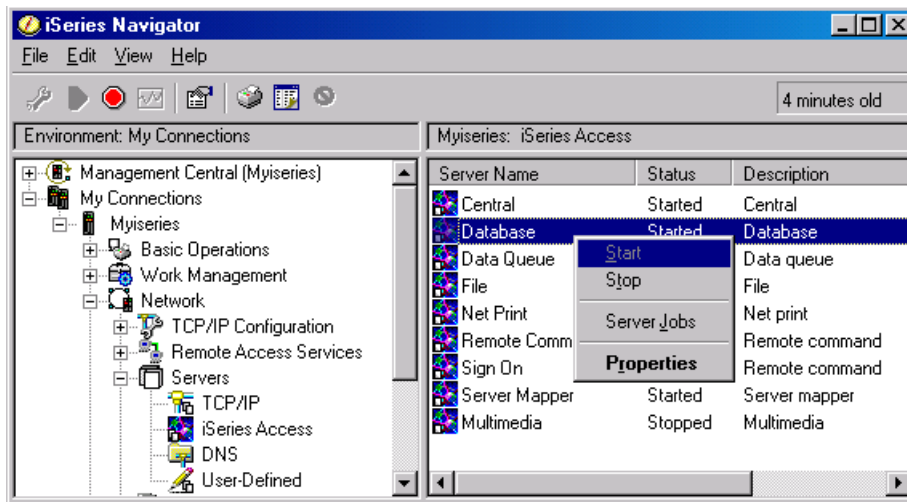


Figure 4-2 iSeries Access for Windows servers

- To verify that a host server prestart job is active, you can look at its status from the iSeries Navigator window as shown in the previous item. Alternatively, you can use the **cwbping** command from a PC command prompt:

```
cwbping myiSeries
```

The cwbping command is located in the folder where you installed iSeries Access for Windows. If your PC's path does not point to this folder, you may need to navigate into that folder in order for the cwbping command to run.

- For more information about optimized host servers, go the iSeries Information Center at:

<http://www.iseries.ibm.com/infocenter>

Choose your region and version if asked, then select **Connecting to iSeries** → **iSeries Access** → **iSeries Access for Windows**. In the right-side frame, click **Administration** → **Host server administration**.

4.2.3 Supported features

The IBM.Data.DB2.iSeries provider includes the following features:

- Classes to implement ADO.NET interfaces, including:
 - iDB2Connection
 - iDB2Command
 - iDB2Parameter and iDB2ParameterCollection
 - iDB2DataReader
 - iDB2DataAdapter
 - iDB2CommandBuilder
 - iDB2Transaction
 - iDB2Exception, iDB2Error, and iDB2ErrorCollection
 - iDB2Trace
 - iDB2ProviderSettings

Note: The iDB2ProviderSettings class was added to the provider in V5R3M0 service pack SI15176.

► Provider-specific data types that correspond to most iSeries data types:

- iDB2BigInt
- iDB2Binary
- iDB2Blob
- iDB2Char
- iDB2CharBitData
- iDB2Clob
- iDB2Date
- iDB2DbClob
- iDB2Decimal
- iDB2Double
- iDB2Graphic
- iDB2Integer
- iDB2Numeric
- iDB2Real
- iDB2Rowid
- iDB2SmallInt
- iDB2Time
- iDB2TimeStamp
- iDB2VarBinary
- iDB2VarChar
- iDB2VarCharBitData
- iDB2VarGraphic

Note: Large Object data types (iDB2Blob, iDB2Clob, and iDB2DbClob) were added to the provider in V5R3M0 Service Pack SI15176.

► Provider-specific exceptions:

- iDB2CommErrorException
- iDB2ConnectionFailedException
- iDB2ConnectionTimeoutException
- iDB2ConversionException
- iDB2DCFunctionErrorException
- iDB2ExitProgramErrorException
- iDB2HostErrorException
- iDB2InvalidConnectionStringException
- iDB2MaximumPoolSizeExceededException
- iDB2SQLException
- iDB2SQLParameterErrorException
- iDB2UnsupportedHostVersionException

► Provider-specific ConnectionString properties to enable you to better manage your connection to the iSeries database:

- ConnectionTimeout
- CheckConnectionOnOpen
- Database
- DataCompression
- DataSource
- DefaultCollection
- DefaultIsolationLevel
- HexParserOption
- LibraryList
- MaximumDecimalPrecision
- MaximumDecimalScale

- MaximumInlineLobSize
- MaximumPoolSize
- MaximumUseCount
- MinimumDivideScale
- MinimumPoolSize
- Naming
- Password
- Pooling
- QueryOptionsFileLibrary
- SortLanguageId
- SortSequence
- SortTable
- SSL
- Trace
- UserID

Note: The LibraryList and Naming properties were added to the provider in V5R3M0 Service Pack SI15176. The CheckConnectionOnOpen property was added to the provider in V5R3M0 Service Pack SI17742.

These connection properties are covered in more detail in 4.5.2, “iDB2Connection and ConnectionString properties” on page 48.

4.2.4 Unsupported features

Although the IBM.Data.DB2.iSeries .NET provider supports most data types and features that other DB2 UDB for iSeries data providers support, it has some limitations. The following features are not supported as of this writing:

- ▶ Datalink data type: The provider does not have a datalink data type, although you can use datalinks if you write special code. See 4.7.5, “Using DataLinks” on page 141 for information about how you can read and write datalink values using the provider.
- ▶ User-defined types (UDTs): Although some features may work, extensive testing has not been done using the IBM.Data.DB2.iSeries provider with UDTs.
- ▶ SQL packages (extended dynamic): For SQL packages, use the System.Data.OleDb .NET provider to bridge to the IBMDASQL OLE DB provider included with iSeries Access for Windows, V5R3M0 and later; or use the Microsoft ODBC .NET provider to bridge to the iSeries Access for Windows ODBC driver.
- ▶ Record level access: For record level access, you can use the IBMDA400 or IBMDARLA OLE DB provider included with iSeries Access for Windows. Testing of record level access through the System.Data.OleDb bridge has not been performed.
- ▶ Data queues: For data queues, you can use the IBMDA400 OLE DB provider included with iSeries Access for Windows. Testing of data queues through the System.Data.OleDb bridge has not been performed.
- ▶ Remote command and program call: For remote command and remote program call, you can use the IBMDA400 OLE DB provider included with iSeries Access for Windows. Testing of remote command and program call through the System.Data.OleDb bridge has not been performed. If your remote command or program does not contain any output parameters, you may be able to run the command through IBM.Data.DB2.iSeries using QCMDEXC. See 4.6.5, “Calling a program or CL command using QCMDEXC” on page 120 for more information. Alternatively, you can wrap your command or program using a stored procedure.

- ▶ COM+, including Microsoft Distributed Transaction Coordinator (DTC) and Microsoft Transaction Services (MTS): For Microsoft distributed transaction coordinator, you can use the System.Data.OleDb .NET provider to bridge to the IBMDASQL OLE DB provider included with iSeries Access for Windows, V5R3M0 and later; or use the Microsoft ODBC .NET provider to bridge to the iSeries Access for Windows ODBC driver.
- ▶ Distributed relational database architecture (DRDA®), including the CONNECT and DISCONNECT statements: For DRDA, you can use the Microsoft ODBC .NET provider to bridge to the iSeries Access for Windows ODBC driver.
- ▶ SET TRANSACTION, COMMIT, and ROLLBACK statements: Instead, we recommend using the built-in transaction support provided via the iDB2Connection.BeginTransaction() method, and the iDB2Transaction object.
- ▶ SET PATH statement: Instead, use the LibraryList property in your iDB2Connection's ConnectionString.
- ▶ SET SCHEMA statement: Instead, use the DefaultCollection property in your iDB2Connection's ConnectionString.

Note: Some operations may work, but record level access, data queues, and remote command and program call have not been tested through the System.Data.OleDb bridge.

4.3 Before we begin

Before we begin writing sample code for the IBM.Data.DB2.iSeries provider, we have to set up the PC and iSeries host.

4.3.1 PC setup

The following software components must be installed onto your PC:

- ▶ Microsoft .NET Framework and Microsoft Visual Studio .NET: Visual Studio .NET will install a version of the .NET Framework if a required version is not already installed. The IBM.Data.DB2.iSeries provider does not require a particular version of the .NET Framework; as of this writing, it has been tested with .NET Framework V1.0 and V1.1.
- ▶ iSeries Access for Windows (5722-XE1), V5R3M0 or later: Be sure to select the .NET **Data Provider** Data Access component, and the **Headers, Libraries, and Documentation** Toolkit component as shown in Figure 4-3 on page 40.

The Data Access .NET Data Provider component containing the runtime IBM.Data.DB2.iSeries provider: The Toolkit Headers, Libraries, and Documentation component contains the Technical Reference for the provider, which contains descriptions of all classes for the IBM.Data.DB2.iSeries provider and some short coding samples.

Tip: You can use iSeries Access for Windows Selective Setup to add or remove components after your initial install.

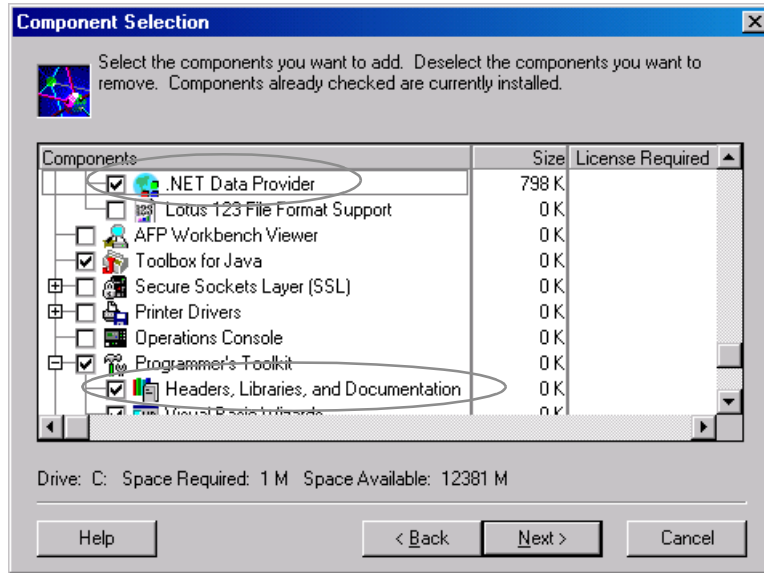


Figure 4-3 iSeries Access for Windows install window

- Applicable service packs or fix packs. We recommend that you always run with the latest iSeries Access for Windows service pack. Service packs can be downloaded and installed by your administrator, or by selecting **Service Packs (Fixes)** from the iSeries Access for Windows Web site at:

<http://www.ibm.com/servers/eserver/iseries/access/windows/>

After you apply a service pack, be sure to reboot your PC if asked to do so. Some new files are not installed properly until after a reboot occurs (for example, if the file is locked by a running application).

The rest of this chapter assumes that your iSeries Access for Windows is at least V5R3M0 with service pack level SI15176.

4.3.2 Host setup

In addition to the PC setup, you need the following host setup:

- Access to an iSeries host. Check the iSeries Access for Windows Web site for supported host versions:
<http://www.ibm.com/servers/eserver/iseries/access/windows/>
- The database host server prestart jobs must be active on your iSeries. See 4.2.2, “Host server jobs” on page 35 for more information about the host server jobs.
- The examples in this chapter assume that you have already created a SQL sample schema on your iSeries called `samp1edb`. The `samp1edb` schema includes many different types of tables used in our coding examples. See 1.4, “DB2 UDB for iSeries sample schema” on page 8 for information about setting up the sample schema.
- Applicable host server PTFs or cumulative PTF levels. To help prevent problems that have already been fixed, you should apply server-side fixes on a regular basis.

4.4 Getting started

Now that you have set up your PC and iSeries, you can start. In this section, we show how to prepare to write an application to use the IBM.Data.DB2.iSeries provider.

4.4.1 Displaying the technical reference

The IBM DB2 UDB for iSeries .NET provider Technical Reference is an important tool when writing your application. It contains all of the class, data type, and exception definitions that are available from the provider. It also contains some coding examples and additional information to help you get started. To display the Technical Reference:

1. From the Windows desktop, select **Start** → **Programs** → **IBM iSeries Access for Windows** → **Programmer's Toolkit** → **Programmer's Toolkit**.
2. In the left pane of the Programmer's Toolkit window, select **Database** → **.NET Framework Classes** (Figure 4-4).
3. In the right pane of the Programmer's Toolkit window, select the **IBM DB2 UDB for iSeries .NET provider Technical Reference** link.

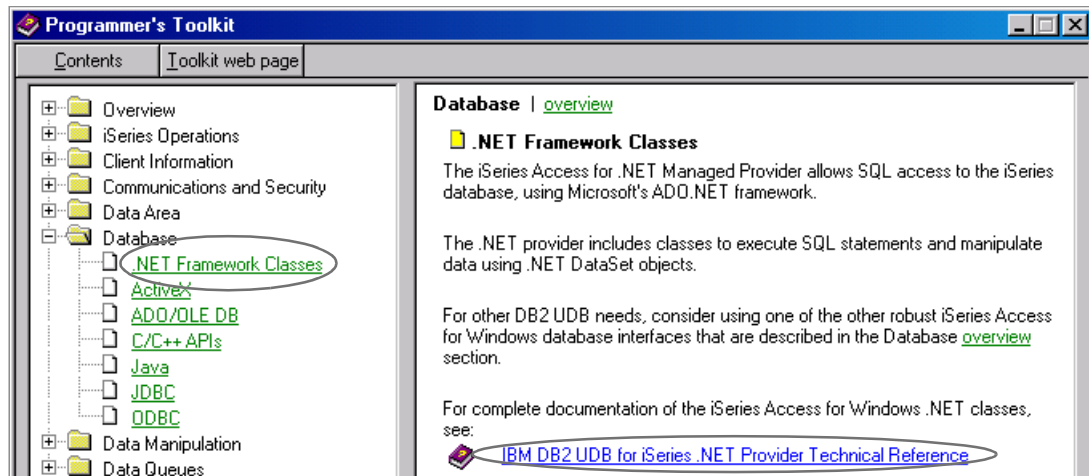


Figure 4-4 iSeries Access for Windows Programmer's Toolkit

The .NET provider Technical Reference (Figure 4-5 on page 42) opens. Keep this window open if you want more information about any of the classes, methods, or properties that are supported by the IBM.Data.DB2.iSeries provider.

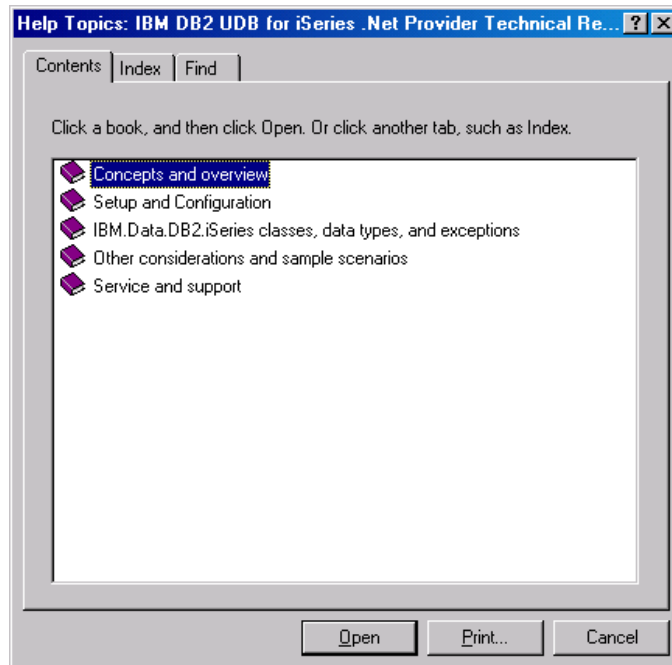


Figure 4-5 IBM DB2 UDB for iSeries .NET provider Technical Reference

Tip: Because you might refer to the Technical Reference on many occasions, you can create a shortcut that points to the location of the Technical Reference .hlp file, for example `c:\Program Files\IBM\Client Access\mri2924\cwbmptch.hlp`. The actual location of the .hlp file may differ depending on your installation.

4.4.2 Starting Visual Studio .NET

To begin using the IBM.Data.DB2.iSeries provider, start Visual Studio .NET and create a new Visual C# project. Most of the examples in this chapter use a Console application, so when you see coding examples, you can assume that they are written to work with a Console application unless we tell you to use a Windows application. For now, we use a Console application, as shown in Figure 4-6 on page 43.

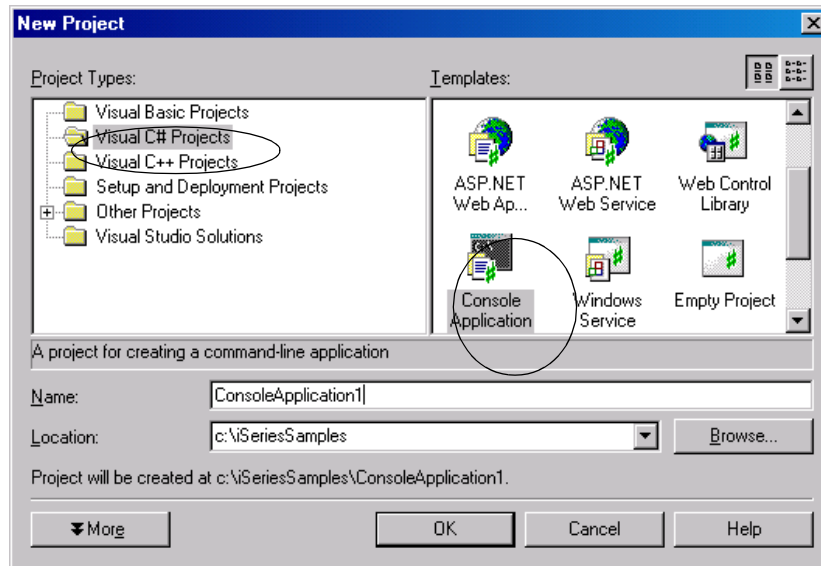


Figure 4-6 Create a C# Console Application

Note: The examples in this chapter use Visual C#. Most are available to download in both Visual Basic and in C#. See Appendix B, “Additional material” on page 261 for information about downloading the samples. Appendix A, “Sample programs” on page 257 contains a list of all the sample programs.

4.4.3 Adding an assembly reference to the provider

Before your application can use the IBM.Data.DB2.iSeries provider classes, you must add an assembly reference so the .NET runtime knows where to find them.

1. From Visual Studio .NET, select **View** → **Solution Explorer** as shown in Figure 4-7.

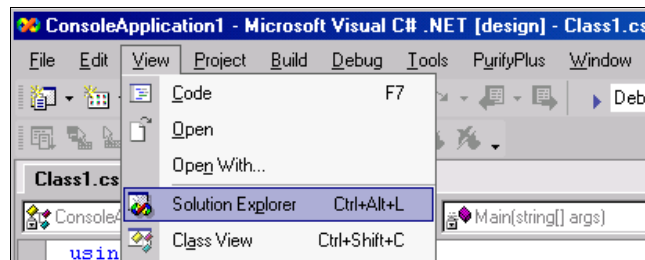


Figure 4-7 Select Solution Explorer

2. In the Solution Explorer (Figure 4-8), right-click **References** and select **Add Reference**.

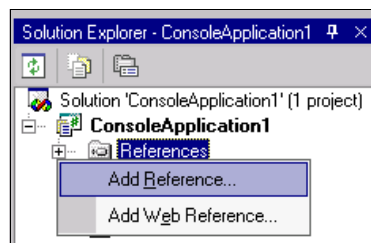


Figure 4-8 Solution Explorer

3. The Add Reference dialog box opens, containing a list of .NET components your application can use, as shown in Figure 4-9. From here:
 - a. Select **IBM DB2 UDB for iSeries .NET provider** and click **Select**.

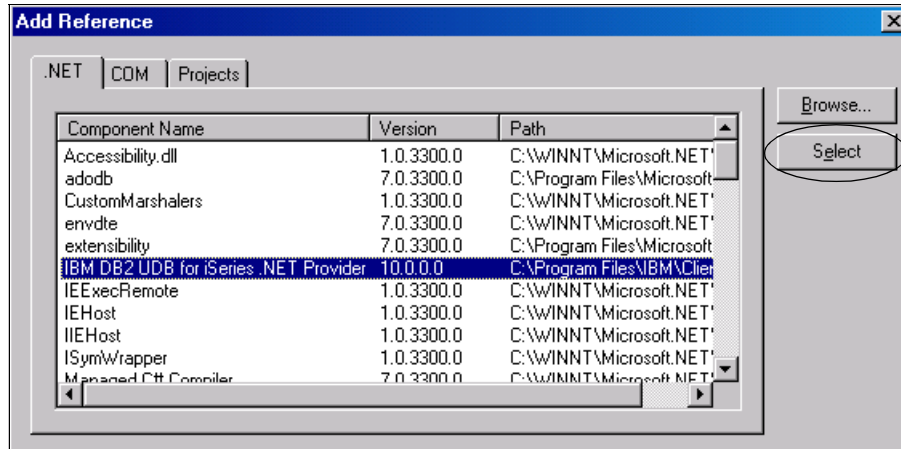


Figure 4-9 Select the IBM DB2 UDB for iSeries .NET provider

- b. Under Selected Components, select **IBM DB2 UDB for iSeries .NET provider** and click **OK** as shown in Figure 4-10.

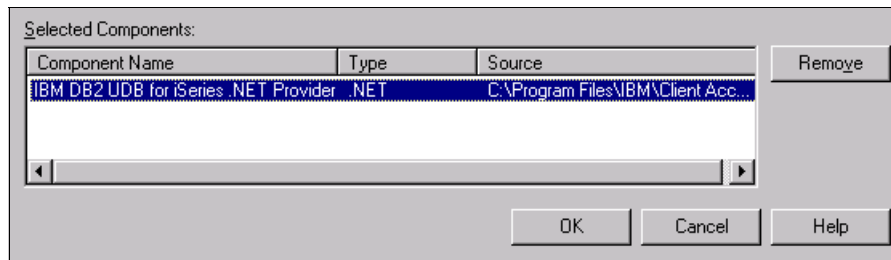


Figure 4-10 Select the IBM DB2 UDB for iSeries .NET provider again

Your application now has an assembly reference to the IBM.Data.DB2.iSeries .NET provider.

4.4.4 Adding a namespace directive

Finally, add a *namespace directive* for easier use of IBM.Data.DB2.iSeries classes. A namespace directive enables you to refer to class names without having to fully qualify them. For example, without the namespace directive, to use an `iDB2Connection` object, type:

```
IBM.Data.DB2.iSeries.iDB2Connection
```

By adding a namespace directive, you can refer to the object directly:

```
iDB2Connection
```

To add a namespace directive using C#, add a *using* directive to your C# source file:

```
using IBM.Data.DB2.iSeries;
```

To add a namespace directive using Visual Basic .NET, add an *Imports* statement to your Visual Basic source file:

```
Imports IBM.Data.DB2.iSeries
```

Your application is now ready to use the IBM.Data.DB2.iSeries .NET provider.

4.5 Provider basics

In this section, we explain some ADO.NET basics as they apply to the IBM.Data.DB2.iSeries .NET provider, including coding examples and insight into some features of this provider.

4.5.1 A simple connection example

Now you can make a connection to the iSeries. Our simple connection example shows how to connect to the iSeries with only a few lines of code.

Write the code

Copy the code from Example 4-1 into your Console application, substituting the name of your iSeries server for *myiSeries* in the example.

Example 4-1 A simple iDB2Connection example

```
using System;
using IBM.Data.DB2.iSeries;

namespace SimpleConnectionExample
{
    /// <summary>
    /// A simple iDB2Connection example
    /// </summary>
    class Class1
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main(string[] args)
        {
            iDB2Connection cn = new iDB2Connection("DataSource=myiSeries;");
            cn.Open();
            Console.WriteLine(cn.JobName);
            cn.Close();
        }
    }
}
```

Run the first three statements of the connection example

Now run the small sample program we just created:

1. From Visual Studio .NET, save your application (**File** → **Save All**).
2. Build the application using **Build** → **Rebuild solution**.
3. Place your cursor on the line containing the text `cn.Close();`
4. Right-click and select **Run To Cursor**. This causes the first three statements to be executed, and the `cn.Close()` statement is highlighted. At this point, you should have an open connection to the iSeries.

Note: The first time you connect, you may see a logon window. Enter your iSeries user ID and password, and click **OK**.

Examine the host job log

The JobName property corresponds to the iSeries host server job that processes requests for your connection. Using this property makes it easier for you to find the host server job, which is handy during problem determination.

When you run the connection example program, a window opens, and the connection's JobName property is printed; for example:

123456/QUSER/QZDASOINIT

Note: Your job number may be different from the one shown here. If you are using a secure connection (SSL property), then your job name will be QZDASSINIT.

Before you let the program continue, look at the host server job that corresponds to the JobName property printed by the connection example program. To look at the server job using iSeries Navigator:

1. From the Windows desktop, select **Start** → **Programs** → **IBM iSeries Access for Windows** → **iSeries Navigator**.
2. From the iSeries Navigator window, expand **My Connections** → *myiSeries* → **Work Management**, substituting the name of your iSeries for *myiSeries*.
3. Right-click **Server Jobs**, and select **Customize this View** → **Include** (Figure 4-11).

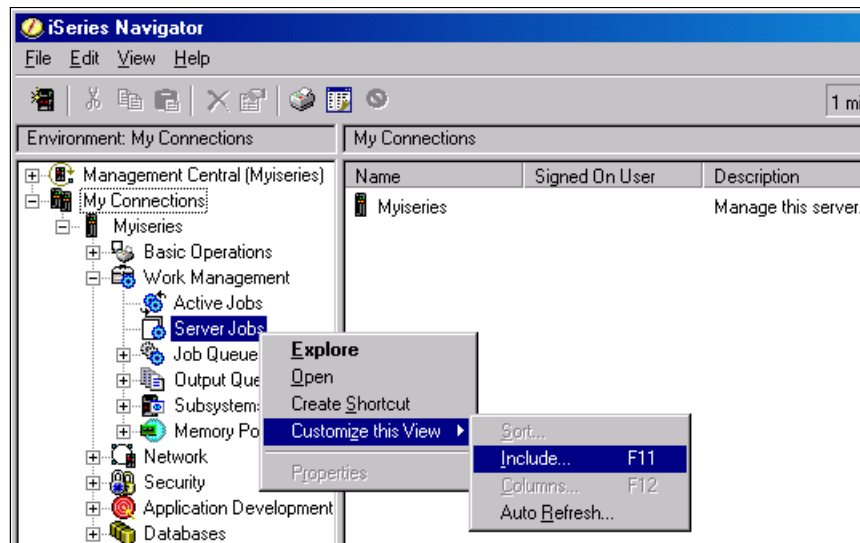


Figure 4-11 Customize the iSeries Navigator Server Jobs view

4. Enter the server job information from the JobName your program printed, as in Figure 4-12 on page 47, and click **OK**. The right pane of your iSeries Navigator window is updated with the job you selected.

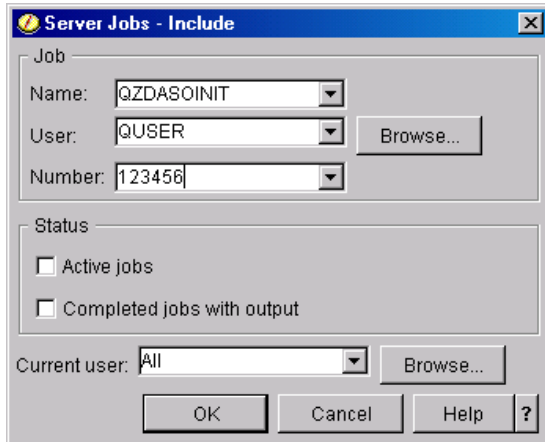


Figure 4-12 Enter the server job information

5. Right-click the QZDASOINIT server job and select **Job Log** (Figure 4-13).

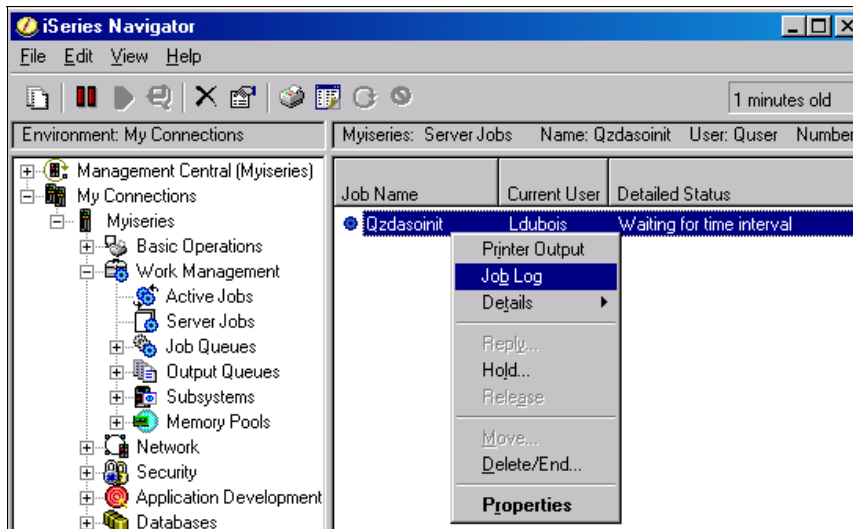


Figure 4-13 Select the job log

The Job Log window opens as in Figure 4-14.

Message ID	Message	Sent	Type
CPIAD02	User LDUBOIS from client 9.10.111.46 connected to server.	11/12/04 2:39:57 PM	Informa
CPF1124	Job 271623/QUSER/QZDASOINIT started on 11/11/04 at 1...	11/11/04 5:15:49 PM	Informa

Figure 4-14 iSeries Navigator Job Log window

Hint: The JobName property is returned in the same format that WRKJOB uses, so if you prefer to use a 5250 workstation or emulator, you can copy and paste the JobName directly from your Console window to your WRKJOB statement; for example:

```
WRKJOB JOB(123456/QUSER/QZDASOINIT)
```

End the program

Return to Visual Studio .NET and press F5 to let the program close the connection and run to completion.

4.5.2 iDB2Connection andConnectionString properties

The connection object for IBM.Data.DB2.iSeries is called *iDB2Connection*. The iDB2Connection object implements the System.Data.IDbConnection interface. You must have a connection object before performing any SQL operations on the iSeries using the .NET provider. You typically create an iDB2Connection object, set the ConnectionString property, open the connection, perform your work using commands, and then close the connection.

Each ADO.NET provider has its own unique set of attributes for initializing the connection. While this makes it harder to write code to easily switch among different providers, it makes sense because each provider “talks” to a different database. In this section, we explain how to create an iDB2Connection object and initialize its ConnectionString.

Creating an iDB2Connection object

To create an iDB2Connection object, use the *new* operator. You can either create an empty connection object and later initialize its ConnectionString, or you can create the object and initialize its ConnectionString in one operation. Example 4-2 shows the two different ways to create a connection.

Example 4-2 iDB2Connection: new operator

```
// Method 1: Create an empty connection and manually set the ConnectionString.
iDB2Connection cn1 = new iDB2Connection();
cn1.ConnectionString = "DataSource=myiSeries;"
```

```
// Method 2: Create a connection and initialize the ConnectionString
// in a single operation. This is a more efficient method.
iDB2Connection cn2 = new iDB2Connection("DataSource=myiSeries;"
```

Setting the ConnectionString property

The iDB2Connection object has only a single writable property: ConnectionString. This property tells the provider which iSeries server to connect to, which schema to use as the default schema, which user ID and password you want to authenticate with, and more. Example 4-1 on page 45 shows how to quickly and easily create a connection to the iSeries using the IBM.Data.DB2.iSeries .NET provider. The example creates an iDB2Connection object and initializes its ConnectionString property using only the DataSource attribute:

```
"DataSource=myiSeries;"
```

The ConnectionString supports many more attributes for better control over managing a connection. In this section, we explain these ConnectionString options in more detail, and include sample ConnectionStrings to illustrate how you can use each option.

Read-only connection properties reflect ConnectionString attributes

Every attribute you set in the ConnectionString corresponds to an iDB2Connection property. All connection properties except the ConnectionString are read-only. After your application sets the ConnectionString property, the provider updates the iDB2Connection object's properties to correspond to these attributes.

Format of the ConnectionString

Items in the ConnectionString appear as name=value pairs, with each pair separated by a semicolon. In this way, you can string together many different attributes in your ConnectionString; for example:

```
"DataSource=myiSeries; DefaultCollection=sampledb; Naming=SQL; LibraryList=sampledb,
*USRLIBL;"
```

ConnectionString properties explained

In this section we describe the properties you can use in your ConnectionString. Information about most of these properties is in the iSeries Access for Windows .NET provider Technical Reference. (See 4.4.1, “Displaying the technical reference” on page 41.)

Authentication on the iSeries

The ConnectionString must *always* include at least the DataSource property, which contains the name or IP address of the iSeries server you want to connect to. To open a connection to the specified iSeries data source, sign on. This is normally done with a user ID and password, but you can also use Kerberos to authenticate. There are two ways to specify sign-on information for use with the IBM.Data.DB2.iSeries .NET provider:

- ▶ Configure the data source using iSeries Navigator.
- ▶ Specify the user ID or password (or both) in your ConnectionString. Applications that run on a Web server, such as the Microsoft Internet Information Services (IIS), should *always* include both the user ID and the password in the ConnectionString to ensure that a logon window is not displayed on the Web server PC. See Chapter 7, “ASP .NET scenario (Web forms)” on page 237 for more about using IBM.Data.DB2.iSeries in a Web environment.

DataSource

The DataSource property is always a required element of the ConnectionString. It specifies the name or IP address of the iSeries you want to connect to. The DataSource connection name is resolved using the iSeries Access for Windows communication component.

Example 4-3 shows how you can use the DataSource property in your ConnectionString.

Example 4-3 Specifying the DataSource property in the ConnectionString

```
iDB2Connection cn = new iDB2Connection();  
cn.ConnectionString = "DataSource=myiSeries;"
```

UserID

The UserID property specifies the user ID for logging on to the iSeries. If UserID is not specified in the ConnectionString, then the iSeries Access for Windows configuration default is used, and iSeries Access for Windows may display a logon prompt when the connection is opened.

Example 4-4 shows how you can use the UserID property in your ConnectionString.

Example 4-4 Specifying the UserID property in the ConnectionString

```
iDB2Connection cn = new iDB2Connection();  
cn.ConnectionString = "DataSource=myiSeries; UserID=myuserid;"
```

Password

The Password property specifies the iSeries password used for logging onto the iSeries. If Password is not specified in the ConnectionString, then iSeries Access for Windows may display a logon prompt when the connection is opened.

Example 4-5 shows how you can use the Password property in your ConnectionString.

Example 4-5 Specifying the Password property in the ConnectionString

```
iDB2Connection cn = new iDB2Connection();  
cn.ConnectionString = "DataSource=myiSeries; UserID=myuserid; Password=mypassword;"
```

Using Kerberos authentication

Most applications connect to the iSeries using the UserID and Password connection properties, but some applications prefer to use Kerberos authentication. Before opening a connection using Kerberos, you must use iSeries Navigator to define the connection and configure the connection for Kerberos. In your `ConnectionString`, do not specify either the UserID or the Password property; the connection will use the configured default of Kerberos.

For more information about using Kerberos with iSeries Access for Windows, see the IBM Redbook *iSeries Access for Windows V5R2 Hot Topics: Tailored Images, Application Administration, SSL, and Kerberos*, SG24-6939. You can read more about Kerberos in the iSeries Access for Windows User's Guide. To display the User's Guide from the Windows desktop, select **Start** → **Programs** → **IBM iSeries Access for Windows** → **User's Guide**. In the User's Guide, click the **Index** tab and type Kerberos.

SSL

The SSL property enables you to connect to the iSeries using an SSL (Secure Sockets Layer) connection. If SSL is not specified in the `ConnectionString`, then the iSeries Access for Windows configuration default will be used.

To connect using SSL, first you must install and configure the SSL component of iSeries Access for Windows. If you try to open a connection using SSL when SSL is not installed and configured on your PC, you will get an `iDB2CommErrorException`.

To verify that the host server connection is working properly with SSL configured, start a PC command prompt, and type (substituting the name of your iSeries for *iSeries*):

```
cwbping iSeries /ssl:1
```

For more about using SSL with iSeries Access for Windows, see the IBM Redbook *iSeries Access for Windows V5R2 Hot Topics: Tailored Images, Application Administration, SSL, and Kerberos*, SG24-6939. Read more about SSL in the iSeries Access for Windows User's Guide. From the Windows desktop, select **Start** → **Programs** → **IBM iSeries Access for Windows** → **User's Guide**. In the User's Guide, click on the **Index** tab and type SSL.

Example 4-6 shows how you can use the SSL property in your `ConnectionString`.

Example 4-6 Specifying the SSL property in the ConnectionString

```
iDB2Connection cn = new iDB2Connection();  
cn.ConnectionString = "DataSource=myiSeries; SSL=true;";
```

ConnectionTimeout

The `ConnectionTimeout` property can be used to specify the longest time (in seconds) to wait for the connection to open. A value of 0 means that the connection will wait indefinitely. In some cases, the default value of 30 (seconds) may not be long enough; for example, if your communication link is especially slow. If `ConnectionTimeout` is not specified in the `ConnectionString`, then the iSeries Access for Windows configuration default will be used.

Example 4-7 shows how to use the `ConnectionTimeout` property in your `ConnectionString`.

Example 4-7 Specifying the ConnectionTimeout property in the ConnectionString

```
iDB2Connection cn = new iDB2Connection();  
cn.ConnectionString = "DataSource=myiSeries; ConnectionTimeout=60;";
```

Database

One of the outstanding features of the iSeries is its database. Unlike some other platforms, the DB2 UDB for iSeries is fully integrated with the operating system. Because of this, the IBM.Data.DB2.iSeries .NET provider does not require you to specify a Database property when you connect. Instead, the provider uses the Database property to enable you to connect to an independent auxiliary storage pool (IASP), which is sometimes referred to as a catalog, database, relational database, or RDB.

In most cases, you should *not* specify the Database in the ConnectionString. The only time you should use Database is when you want your application to connect to an IASP other than the system default *SYSBAS. The only way your application can reference a schema that resides in an IASP is to connect to that IASP via the Database connection property.

If you do not specify the Database in the ConnectionString, the database will be determined by the job description. After a connection is opened, the provider updates the Database property to reflect the database that was connected to.

Example 4-8 shows how you can use the Database property in your ConnectionString.

Example 4-8 Specifying the Database property in the ConnectionString

```
iDB2Connection cn = new iDB2Connection();  
cn.ConnectionString = "DataSource=myiSeries; Database=myIASP;"
```

Hint: You can determine the RDB (Database) name for your iSeries by using the WRKRDBDIRE command. The system database is indicated by a Remote Location of *LOCAL.

Naming, DefaultCollection, LibraryList, and unqualified object names

These properties control how the iSeries server resolves unqualified object names such as tables, external programs, and stored procedures. *Unqualified object names* refer to an object without specifying the schema the object resides in:

- ▶ Example of a qualified object name: `samp1edb.employee` or `samp1edb/employee`
- ▶ Example of an unqualified object name: `employee`

When the iSeries server is given a request, it must determine which schema that object resides in. For qualified object names, the schema is included in the request. For unqualified object names, the iSeries server must use a different method to determine the schema. Some objects (such as tables, external programs, and views) are resolved using the concept of a *default schema*, but other objects (such as stored procedures) use the *library list* to resolve unqualified object names. This topic is discussed in greater detail in the DB2 Universal Database for iSeries SQL Reference, which you can find in the iSeries Information Center at:

<http://www.iseries.ibm.com/infocenter>

Select **Database** → **Reference** → **SQL Reference** → **Language elements** → **Naming conventions**.

Naming

The Naming property is used to set the iSeries naming convention used for SQL requests. It affects how the host server resolves unqualified object names, and it can also affect how the DefaultCollection property is used (see “DefaultCollection” on page 52).

- ▶ When Naming is set to SQL (the default), object names containing a schema use a period (.) to separate the schema name from the object name; for example:

`schema.object`

- ▶ When Naming is set to System, object names containing a schema use a forward slash (/) to separate the schema name from the object name; for example:

schema/object

Note: The Naming property was added to iSeries Access for Windows V5R3M0 in service pack SI15176.

Example 4-9 shows how you can use the Naming property in your ConnectionString.

Example 4-9 Specifying the Naming property in the ConnectionString

```
iDB2Connection cn = new iDB2Connection();  
cn.ConnectionString = "DataSource=myiSeries; Naming=System;";
```

DefaultCollection

The DefaultCollection property corresponds to the default schema that is used to resolve some unqualified object names such as tables and external programs.

If your application sets the DefaultCollection property in the ConnectionString, then that becomes the default schema for your connection and is used to resolve many unqualified object types, for both SQL and System naming.

If the application does *not* set the DefaultCollection property, then the behavior depends on the naming convention in use:

- ▶ With SQL naming, the provider sets the DefaultCollection property to the user ID that opened the connection. This reflects the iSeries server's notion of a *run-time authorization identifier*.
- ▶ With System naming, the provider does not change the DefaultCollection property after a connection is opened. In this case, the iSeries server resolves unqualified object names using the host server job's library list.

Example 4-10 shows how to use the DefaultCollection property in your ConnectionString.

Example 4-10 Specifying the DefaultCollection property in the ConnectionString

```
iDB2Connection cn = new iDB2Connection();  
cn.ConnectionString = "DataSource=myiSeries; DefaultCollection=sampledb;";
```

LibraryList

On the iSeries server, each job has a library list associated with it. Each job's library list is composed of several different parts, including the server portion of the library list, the user portion of the library list, product libraries, and the current library.

The LibraryList property enables you to set the user portion of the library list for the iSeries host server job. LibraryList is a list of schema names separated by commas. The list can include the special schema name *USRLIBL, which is a placeholder for the library list associated with the user profile of the current host server job. By placing other schema names before or after *USRLIBL, your application can control the order in which these schemas are referenced. If *USRLIBL is not included in the LibraryList, the library list associated with the user profile of the current host server job is *replaced*. To add to the library list, instead of replacing it, your application must specify *USRLIBL somewhere in the LibraryList.

The Database host server always includes the user profile's *current library* setting in the job library list.

The provider does not automatically include the DefaultCollection in the library list. If you want the DefaultCollection added to the library list, you must set it using the LibraryList property.

One reason to use the LibraryList property is when you want to test some SQL objects before they are ready to go into production. You can create the test objects into a test schema and add this test schema before your production schema in the LibraryList. When your testing is complete, you can move the test objects into your production schema and remove the test schema from your LibraryList. Another reason to use LibraryList is when you want different users to use different schemas. This way, you need only a single line of code to change which schema your application uses.

Note: The LibraryList property was added to iSeries Access for Windows V5R3M0 in service pack SI15176.

Example 4-11 shows how you can use the LibraryList property in your ConnectionString. Before running this example, perform the following setup on your iSeries host:

1. Create three schemas called SCHEMA1, SCHEMA2, and SCHEMA3:

```
CREATE COLLECTION SCHEMA1
CREATE COLLECTION SCHEMA2
CREATE COLLECTION SCHEMA3
```

2. Create a table called TEST in each of the three schemas:

```
CREATE TABLE SCHEMA1/TEST (CHAR1 CHAR(20))
CREATE TABLE SCHEMA2/TEST (CHAR1 CHAR(20))
CREATE TABLE SCHEMA3/TEST (CHAR1 CHAR(20))
```

3. Insert different data into each of the three tables:

```
INSERT INTO SCHEMA1/TEST VALUES('Data from SCHEMA1')
INSERT INTO SCHEMA2/TEST VALUES('Data from SCHEMA2')
INSERT INTO SCHEMA3/TEST VALUES('Data from SCHEMA3')
```

Example 4-11 Specifying the LibraryList property in the ConnectionString

```
iDB2Connection cn = new iDB2Connection();
iDB2Command cmd = cn.CreateCommand();
String dataFromTable;

// We have not specified the schema for the TEST table here.
// The schema that will be used will depend upon the
// LibraryList setting.
cmd.CommandText = "select char1 from test";

// Set the LibraryList so SCHEMA1 appears first.
// Read the data from the TEST table.
// Because SCHEMA1 is first in the library list, we should
// see the data from SCHEMA1/TEST.
cn.ConnectionString = "DataSource=myiSeries; Naming=System; LibraryList=SCHEMA1, SCHEMA2,
SCHEMA3, *USRLIBL;";
cn.Open();
dataFromTable = (String)cmd.ExecuteScalar();
Console.WriteLine("    This should be data from SCHEMA1: " + dataFromTable);
cn.Close();

// Set the LibraryList so SCHEMA2 appears first.
// Read the data from the TEST table.
// Because SCHEMA2 is first in the library list, we should
// see the data from SCHEMA2/TEST.
```

```

cn.ConnectionString = "DataSource=myiSeries; Naming=System; LibraryList=SCHEMA2, SCHEMA1,
SCHEMA3, *USRLIBL;";
cn.Open();
dataFromTable = (String)cmd.ExecuteScalar();
Console.WriteLine("    This should be data from SCHEMA2: " + dataFromTable);
cn.Close();

// Set the LibraryList so SCHEMA3 appears first.
// Read the data from the TEST table.
// Because SCHEMA3 is first in the library list, we should
// see the data from SCHEMA3/TEST.
cn.ConnectionString = "DataSource=myiSeries; Naming=System; LibraryList=SCHEMA3, SCHEMA2,
SCHEMA1, *USRLIBL;";
cn.Open();
dataFromTable = (String)cmd.ExecuteScalar();
Console.WriteLine("    This should be data from SCHEMA3: " + dataFromTable);
cn.Close();

// Dispose the command
cmd.Dispose();

```

Connection pooling

The IBM.Data.DB2.iSeries .NET provider supports connection pooling. This support is turned on by default. Connection pooling enables applications that open and close connections frequently to reuse connections so the connections open more quickly. In some cases, you may want to disable connection pooling; for instance, when doing problem determination or when you know your application only opens and closes a single connection once. In most cases, the pooling defaults can be used.

The provider uses the ConnectionString as a key to determining whether a connection can be reused out of the connection pool. For connection pooling to work, the ConnectionString for the connection must be *identical* to a ConnectionString in the pool. If the ConnectionStrings are not identical, then the connection will not be taken from the pool; instead, a new pool will be created and the new connection will be taken from the new pool. Connection pooling is discussed in greater detail in 4.7.6, “Connection pooling” on page 143.

Pooling

Use the Pooling property to turn connection pooling on or off.

Example 4-12 shows how you can use the Pooling property in your ConnectionString.

Example 4-12 Specifying the Pooling property in the ConnectionString

```

iDB2Connection cn = new iDB2Connection();
cn.ConnectionString = "DataSource=myiSeries; Pooling=false;";

```

MaximumPoolSize

In some cases, you may want to limit the number of iSeries host server jobs that can service your .NET application. By default, the provider enables you to create as many connections as you need in a connection pool. With the MaximumPoolSize property, you can limit the pooled connections. If your application tries to open more than the maximum number of connections, an iDB2MaximumPoolSizeExceededException results. A value of -1 (the default) is used to indicate no maximum.

Example 4-13 on page 55 shows how to use the MaximumPoolSize property in your ConnectionString.

Example 4-13 Specifying the `MaximumPoolSize` property in the `ConnectionString`

```
iDB2Connection cn = new iDB2Connection();  
cn.ConnectionString = "DataSource=myiSeries; MaximumPoolSize=5;";
```

MaximumUseCount

Sometimes, due to unforeseen problems, host server jobs can become “stale” and in need of recycling. The `MaximumUseCount` property enables you to specify how many times a pooled connection is used before it is recycled. When the use count reaches this maximum, the provider creates a new pooled connection and deletes the one whose count has reached the maximum. Because creating a new connection is more time-consuming than using a pooled connection, when `MaximumUseCount` is reached you will see a longer delay the first time the new connection is opened.

Example 4-14 shows how to use the `MaximumUseCount` property in your `ConnectionString`.

Example 4-14 Specifying the `MaximumUseCount` property in the `ConnectionString`

```
iDB2Connection cn = new iDB2Connection();  
cn.ConnectionString = "DataSource=myiSeries; MaximumUseCount=100;";
```

MinimumPoolSize

The `MinimumPoolSize` property is used for cases where you know ahead of time that you will likely use a certain minimum number of connections all at once. In this case, you can set `MinimumPoolSize` to that number. The first time you open one of the pooled connections, *all* of the pooled connections make a connection to the iSeries. This means a longer start-up time, and later when the other pooled connections are opened, the startup time will be reduced. When you use `MinimumPoolSize`, the provider always makes sure that *at least* that many pooled connections exist.

Example 4-15 shows how to use the `MinimumPoolSize` property in your `ConnectionString`.

Example 4-15 Specifying the `MinimumPoolSize` property in the `ConnectionString`

```
iDB2Connection cn = new iDB2Connection();  
cn.ConnectionString = "DataSource=myiSeries; MinimumPoolSize=5;";
```

CheckConnectionOnOpen

The `CheckConnectionOnOpen` property can be used to guard against communication errors that result from trying to open a pooled connection whose host server job has ended. These communication errors could occur, for example, if you leave your application running overnight, and the iSeries server is IPLed for maintenance. When your application tries to Open a pooled connection, it gets a communication error because the pooled host server job has ended. The normal recovery for a communication error is to Close and then Open (or re-Open) the failing connection; but when there are potentially many pooled connections whose server jobs have ended, you could still end up with communication errors when the connection is re-opened if you pick up a different broken pooled connection.

Setting `CheckConnectionOnOpen=true` in your `ConnectionString` causes the provider to send a small amount of data to the host server when you Open the connection. If the provider detects that the host server job is not responding, it opens a new connection to the server. Otherwise it returns the pooled connection. Note that in the case of a persistent communication error, you may still see communication errors on Open, even when using the `CheckConnectionOnOpen` property.

Note: The CheckConnectionOnOpen property was added to iSeries Access for Windows V5R3M0 in service pack SI17742.

Example 4-16 shows how you can use the CheckConnectionOnOpen property in your ConnectionString.

Example 4-16 Specifying the CheckConnectionOnOpen property in the ConnectionString

```
iDB2Connection cn = new iDB2Connection();  
cn.ConnectionString = "DataSource=myiSeries; CheckConnectionOnOpen=true;";
```

SortSequence

The iSeries host server enables you to customize how string values are compared or sorted. The sort method used by the host server job processing requests on behalf of your application can be configured by using the SortSequence property. There are four different ways strings can be sorted on the iSeries:

- ▶ Sort based on *hexadecimal values*, where the hexadecimal values of the characters are compared. (This is the default sort method.) You can set this sort method using the SortSequence property as in Example 4-17.

Example 4-17 Specifying SortSequence=Hex property in the ConnectionString

```
iDB2Connection cn = new iDB2Connection();  
cn.ConnectionString = "DataSource=myiSeries; SortSequence=Hex;";
```

- ▶ Sort based on a *shared weight table*, where uppercase and lowercase characters compare and sort the same. To sort this way, you must specify both the SortSequence property and the SortLanguageId property. SortLanguageId is the three-letter language identifier. Example 4-18 shows a ConnectionString that uses a shared weight table.

Example 4-18 Specifying SortSequence=SharedWeight property in the ConnectionString

```
iDB2Connection cn = new iDB2Connection();  
cn.ConnectionString = "DataSource=myiSeries; SortSequence=SharedWeight; SortLanguageId=ENU;";
```

- ▶ Sort based on a *unique weight table*, where uppercase and lowercase characters compare and sort differently. To sort this way, you must specify both the SortSequence property and the SortLanguageId property. SortLanguageId is the three-letter language identifier. Example 4-19 shows a unique weight table.

Example 4-19 Specifying the SortSequence=UniqueWeight property in the ConnectionString

```
iDB2Connection cn = new iDB2Connection();  
cn.ConnectionString = "DataSource=myiSeries; SortSequence=UniqueWeight; SortLanguageId=ENU;";
```

- ▶ Sort based on your own *user-specified preferences*. This sort method requires you to specify both the SortSequence property and the SortTable property. SortTable is the schema and table name of a sort sequence table. For more information, see “SortTable” on page 57. Example 4-20 shows a ConnectionString that specifies a user defined sort sequence table.

Example 4-20 Specifying the SortSequence=UserSpecified property in the ConnectionString

```
iDB2Connection cn = new iDB2Connection();  
cn.ConnectionString = "DataSource=myiSeries; SortSequence=UserSpecified;  
SortTable=myschema/mytable;";
```

SortLanguageId

The SortLanguageId property is used only when SortSequence is either SharedWeight or UniqueWeight. It is a three-character identifier from a list of languages supported by the system.

Example 4-21 shows how to use the SortLanguageId property in your ConnectionString.

Example 4-21 Specifying the SortLanguageId property in the ConnectionString

```
iDB2Connection cn = new iDB2Connection();  
cn.ConnectionString = "DataSource=myiSeries;SortSequence=SharedWeight;SortLanguageId=ENU;"
```

Tip: The supported language identifiers are the same identifiers that are supported when you use interactive SQL and select Change Session Attributes. To see this list, press F4 from the Change Session Attributes Language identifier field.

SortTable

The SortTable property is used only when SortSequence is UserSpecified. It enables you to define your own sort preferences. For information about creating your own sort sequence table, read about the iSeries CRTTBL command in the iSeries Information Center at:

<http://www.iseries.ibm.com/infocenter>

Select **Programming** → **CL** → **Alphabetic list of commands** → **Create table (CRTTBL) command**.

Example 4-22 shows how you can use the SortTable property in your ConnectionString.

Example 4-22 Specifying the SortTable property in the ConnectionString

```
iDB2Connection cn = new iDB2Connection();  
cn.ConnectionString = "DataSource=myiSeries; SortSequence=UserSpecified;  
SortTable=myschema/mytable;"
```

Note: The SortTable must be in the form schema/table (not schema.table), even when SQL naming convention is being used. Both the schema and table name must be specified, and they can each be up to 10 characters long.

DataCompression

In the ADO.NET environment, your application may send and receive large amounts of data to and from your iSeries server. Because the communications link is often a bottleneck for application performance, it is desirable to reduce the amount of data that must be sent and received over the communications link.

To help alleviate this problem, the IBM.Data.DB2.iSeries provider supports the DataCompression property. Setting DataCompression to true (the default) turns on record length encoding (RLE) compression. RLE compression can significantly reduce the amount of data, and therefore the time, that is needed to send or receive data to and from the iSeries.

Example 4-23 shows how to use the DataCompression property in your ConnectionString.

Example 4-23 Specifying the DataCompression property in the ConnectionString

```
iDB2Connection cn = new iDB2Connection();  
cn.ConnectionString = "DataSource=myiSeries; DataCompression=True;"
```

MaximumInlineLobSize

Large objects (LOBs) are used to hold very large binary or character data. When using LOBs, you run an inherent risk of overloading your system resources (such as memory and disk space). The iSeries server provides two ways to handle LOB data:

- ▶ As *inline* data
- ▶ By using *locators*

With inline LOBs, large data is sent and received as-is in the data stream. For instance, if a large object is 500,000 bytes in size, accessing a record containing that LOB means that the entire 500,000 bytes is sent all at once, regardless of whether your application ever reads the data.

When using LOB locators, only a four-byte locator is sent and received for each large object. The entire large object is read from or written to only when your application requires it.

The `MaximumInlineLobSize` property helps you tune the way you send and receive LOB data by controlling the threshold at which LOB data is transferred using LOB locators, instead of as inline data. LOB data that is smaller than the `MaximumInlineLobSize` is transferred as inline data, and LOB data that is larger than the `MaximumInlineLobSize` is transferred using locators. `MaximumInlineLobSize` is an integer in the range of 0 to 15360 that specifies the maximum size (in kilobytes) of a LOB that can be retrieved from the host server in a single operation.

Example 4-24 shows how you can use the `MaximumInlineLobSize` property in your `ConnectionString`.

Example 4-24 Specifying the `MaximumInlineLobSize` property in the `ConnectionString`

```
iDB2Connection cn = new iDB2Connection();  
cn.ConnectionString = "DataSource=myiSeries; MaximumInlineLobSize=32;";
```

Note: Because a `DataAdapter` always reads all of the query data into memory at once, the `MaximumInlineLobSize` property is most useful when used with a `DataReader`.

For more information about using LOBs with the `IBM.Data.DB2.iSeries` provider, see 4.7.2, “Using large objects (LOBs)” on page 132.

HexParserOption

Use the `HexParserOption` to configure the way you want hexadecimal constants to be treated by the iSeries server. The iSeries deviates from the ISO and ANSI SQL standards, and by default treats hexadecimal constants as character data instead of as binary data. To change this default, set the `HexParserOption` to `Binary`. `HexParserOption` corresponds to the `SQLCURRULE` parameter of the `SET OPTION` statement.

Tip: You cannot mix binary and character data in the same statement if they both use hexadecimal literal strings. The `HexParserOption` (and `SQLCURRULE`) allow hexadecimal constants to be used *either* with character columns *or* with binary columns. When using the `IBM.Data.DB2.iSeries .NET` provider, you can avoid this problem by using parameter markers instead of using hard-coded literal strings. See 4.5.4, “Using parameters in your SQL statements” on page 74 for more information.

Example 4-25 on page 59 shows how to use the `HexParserOption` property in your `ConnectionString`.

Example 4-25 Specifying the HexParserOption property in the ConnectionString

```
iDB2Connection cn = new iDB2Connection();  
cn.ConnectionString = "DataSource=myiSeries; HexParserOption=Binary;";
```

DefaultIsolationLevel

This property is used to set the default isolation level used when a new transaction is started via the iDB2Connection object's BeginTransaction() method call. Figure 4-1 shows how .NET Framework IsolationLevel enumeration values map to the iSeries isolation levels.

Table 4-1 How IBM.Data.DB2.iSeries maps System.Data.IsolationLevel to iSeries isolation levels

System.Data.IsolationLevel enumeration	iSeries isolation level
Chaos	No commit (*NONE)
ReadCommitted	Cursor Stability (*CS)
ReadUncommitted	Uncommitted Read (*CHG)
RepeatableRead	Read Stability (*ALL)
Serializable	Repeatable Read (*RR)

When your application is not in transaction mode (before starting a transaction, and after a transaction has been committed or rolled back), the provider runs with an isolation level of *NONE.

For more information about isolation levels, see the DB2 Universal Database for iSeries SQL Reference, which you can find in the iSeries Information Center at:

<http://www.iseries.ibm.com/infocenter>

Select **Database** → **Reference** → **SQL Reference** → **Concepts** → **Isolation level**.

Example 4-26 shows how to use the DefaultIsolationLevel property in your ConnectionString.

Example 4-26 Specifying the DefaultIsolationLevel property in the ConnectionString

```
iDB2Connection cn = new iDB2Connection();  
cn.ConnectionString = "DataSource=myiSeries; DefaultIsolationLevel=ReadCommitted;";
```

QueryOptionsFileLibrary

The iSeries enables you to use the CHGQRYA command to change attributes used for database queries. You can use the QueryOptionsFileLibrary property to specify a *query options file* library, which contains a query options file called QAQQINI that can be used to control your query options via a trigger program. When you use the QueryOptionsFileLibrary property, the provider issues the following command on behalf of your application:

```
CHGQRYA QRYOPLIB(QueryOptionsFileLibrary)
```

This property should be used with caution, because it can adversely affect your performance. For more information, go to the iSeries Information Center at:

<http://www.iseries.ibm.com/infocenter>

Select **Database** → **Performance and optimization** → **Optimizing query performance using optimization tools** → **Change the attributes of your queries with the Change Query Attributes (CHGQRYA) command** → **Control queries dynamically with the query options file QAQQINI**.

Example 4-27 shows how you can use the QueryOptionsFileLibrary property in your ConnectionString.

Example 4-27 Specifying the QueryOptionsFileLibrary property in the ConnectionString

```
iDB2Connection cn = new iDB2Connection();  
cn.ConnectionString = "DataSource=myiSeries; QueryOptionsFileLibrary=myschema;"
```

Attributes affecting decimal and numeric result data

In its V5R3M0 release, the iSeries enhanced its support for decimal and numeric data types, expanding the maximum precision and scale for these data types from 31 to 63 digits. With these increased maximums, arithmetic operations involving decimal and numeric data can possibly return values with greater precision and scale. The MaximumDecimalPrecision, MaximumDecimalScale, and MinimumDivideScale properties affect how the iSeries server returns data resulting from arithmetic operations on decimal or numeric data. These properties are ignored when the iSeries server version is not V5R3M0 or greater.

MaximumDecimalPrecision

This property is used to configure the largest precision returned by the iSeries server when performing arithmetic operations on decimal and numeric data. MaximumDecimalPrecision can be set to either 31 (the default), or 63, the largest decimal or numeric value supported by the iSeries. The setting affects the size of the result data returned.

Example 4-28 shows how you can use the MaximumDecimalPrecision property in your ConnectionString.

Example 4-28 Specifying the MaximumDecimalPrecision property in the ConnectionString

```
iDB2Connection cn = new iDB2Connection();  
cn.ConnectionString = "DataSource=myiSeries; MaximumDecimalPrecision=63;"
```

MaximumDecimalScale

The MaximumDecimalScale is used to configure the largest scale of decimal or numeric result data returned from arithmetic operations. This value can be in the range of 0 to MaximumDecimalPrecision.

Example 4-29 shows how you can use the MaximumDecimalScale property in your ConnectionString.

Example 4-29 Specifying the MaximumDecimalScale property in the ConnectionString

```
iDB2Connection cn = new iDB2Connection();  
cn.ConnectionString = "DataSource=myiSeries; MaximumDecimalPrecision=63;  
MaximumDecimalScale=63;"
```

MinimumDivideScale

MinimumDivideScale is used to indicate the smallest scale that will be returned for result data when dividing decimal or numeric data. MinimumDivideScale can be any value from 0 to 9, but must not be greater than the MaximumDecimalScale.

Example 4-30 shows how to use the MinimumDivideScale property in your ConnectionString.

Example 4-30 Specifying the MinimumDivideScale property in the ConnectionString

```
iDB2Connection cn = new iDB2Connection();  
cn.ConnectionString = "DataSource=myiSeries;MaximumDecimalScale=20;MinimumDivideScale=5;"
```

MaximumDecimalPrecision and MaximumDecimalScale Example

This example shows how to illustrate these decimal and numeric result data properties, substituting your own schema and table name for *MYSCHEMA.MYTABLE*:

1. Create a table on your iSeries:

```
CREATE TABLE MYSCHEMA.MYTABLE (DEC1 DECIMAL(63, 62))
```

2. Insert a large decimal value into the table:

```
INSERT INTO MYSCHEMA.MYTABLE  
VALUES(1.23456789012345678901234567890123456789012345678901234567890123)
```

3. Copy the code sample from Example 4-31 into a C# Console application.

Example 4-31 MaximumDecimalPrecision and MaximumDecimalScale

```
static void Main(string[] args)
{
    // Input args:
    //     precision = Value to use for MaximumDecimalPrecision (31 or 63)
    //     scale     = Value to use for MaximumDecimalScale (0 to precision)

    // Validate the input args
    if (args.GetUpperBound(0) != 1)
    {
        Console.WriteLine("Valid arguments: MaximumDecimalPrecision MaximumDecimalScale");
        return;
    }
    int prec      = Int32.Parse(args[0]);
    int scale     = Int32.Parse(args[1]);

    if (((prec != 31) && (prec != 63)) ||
        (scale < 0) || (scale > prec))
    {
        Console.WriteLine("Invalid argument. Precision must be 31 or 63, and scale must be 0 to precision.");
        return;
    }

    // Open a connection to the iSeries
    String connectionString = "DataSource=miSeries; MaximumDecimalPrecision=" +
        prec.ToString() + "; " +
        " MaximumDecimalScale=" + scale.ToString() + "; ";

    iDB2Connection cn = new iDB2Connection(connectionString);
    cn.Open();

    // Read the result data as an arithmetic result
    iDB2Command cmd = new iDB2Command("select (dec1/2) from myschema.mytable", cn);
    iDB2DataReader dr = cmd.ExecuteReader();

    // Display the result of the query.
    while (dr.Read())
        Console.WriteLine(dr.GetiDB2Decimal(0).ToString());

    dr.Close();

    // Close the connection
    cn.Close();
}
```

1. Save the application (**File** → **Save All**), and build the application (**Build** → **Rebuild solution**).
2. Run the application, and pass different values for the precision and scale parameters. You will see that the result data you get back changes depending on how you set the `MaximumDecimalPrecision` and `MaximumDecimalScale` properties. Table 4-2 shows some sample outputs from the program.

Table 4-2 Sample outputs from the sample program

Program parameters	Resulting output
63 63	.6172839450617283945061728394506172839450617283945061
63 12	.617283945061
31 31	.6172839450617283945061728394506

Properties that reflect the state of the connection

While most of the `iDB2Connection` properties are a reflection of values you specify in the `ConnectionString`, some properties are set by the `IBM.Data.DB2.iSeries .NET` provider independent of the `ConnectionString`. These properties are `Provider`, `State`, `ServerVersion`, and `JobName`.

Provider

The `Provider` property is a constant string value that always returns the value `IBM.Data.DB2.iSeries`. It is not particularly useful, but is provided so you can programmatically determine the name of the provider.

Example 4-32 shows how you can use the `Provider` property.

Example 4-32 Using the `Provider` property

```
iDB2Connection cn = new iDB2Connection();
cn.ConnectionString = "DataSource=myiSeries;";
Console.WriteLine(cn.Provider);
```

State

The `State` property reflects the state of the connection. Before you open your `iDB2Connection`, and after you close it, the state is `ConnectionState.Closed`. After the connection is successfully opened, the `State` property changes to `ConnectionState.Open`.

Example 4-33 shows how you can use the `State` property.

Example 4-33 Using the `State` property

```
void closeConnection(iDB2Connection cn)
{
    if (cn.State == System.Data.ConnectionState.Open)
        cn.Close();
}
```

ServerVersion

This property reflects the iSeries server version. It is returned as a string of the form `vv.rr.mmmm`, so for example `V5R3M0` is returned as `05.03.0000`. This property can be used if you want to take advantage of iSeries features that are only available starting with a certain release. As an example of this, you can check the `ServerVersion`, and if the server is at least

V5R3M0, then you can use the true Binary data type, which is first available in that release. To check the server version in this manner, try the code sample shown in Example 4-34.

Example 4-34 Using the ServerVersion property

```
iDB2Connection cn = new iDB2Connection();
cn.ConnectionString = "DataSource=myiSeries;";
cn.Open();

// Get the ServerVersion and convert it to an integer
String ver = cn.ServerVersion;
int vvrmmmm = (Int32.Parse(ver.Substring(0, 2)) * 10000) + (Int32.Parse(ver.Substring(3, 2)) * 100) + (Int32.Parse(ver.Substring(6, 4)));
if (vvrmmmm >= 50300)
{
    // Do work that is specific to V5R3M0 and later
}
```

Reminder: Because ServerVersion reflects the iSeries server version, it only contains meaningful information after you open your connection.

JobName

The JobName property is a useful tool to help you determine which iSeries server job services your SQL requests. This property is valid only after the connection is open. Example 4-35 shows how you can use the JobName property.

Example 4-35 Using the JobName property

```
iDB2Connection cn = new iDB2Connection();
cn.ConnectionString = "DataSource=myiSeries;";
cn.Open();
Console.WriteLine("Job Name: " + cn.JobName);
```

ConnectionString attributes used for problem determination

Even when we write “perfect” code, things may not go according to plan, and we must determine the source of the problem. The IBM.Data.DB2.iSeries .NET provider supports a ConnectionString attribute called Trace that enables you to easily turn on various server-side traces. For more about problem determination, see 4.10, “Troubleshooting” on page 166.

Trace

When the Trace property is specified in the ConnectionString, the provider sends commands to the iSeries to enable the specified traces. This property should *only* be used when you are doing problem determination, or when recommended by IBM Service. Turning traces on affects performance and should not be used during normal processing.

Because connection pooling is enabled by default, the Trace property causes traces to be turned on the first time the pooled connection is opened, and turned off only when the pooled connection is released, either when the MaximumUseCount is reached or when the iDB2Connection object is disposed of. Because of this, we recommend that when using any of the Trace options, you turn off connection pooling unless the problem can only be reproduced with connection pooling enabled.

The Trace options are described in the IBM.Data.DB2.iSeries Technical Reference (see 4.4.1, “Displaying the technical reference” on page 41). To turn on several host traces at the same time, put them all in the ConnectionString, separated by commas, as in Example 4-36 on page 64.

Example 4-36 Specifying the Trace property in the ConnectionString

```
iDB2Connection cn = new iDB2Connection();  
cn.ConnectionString = "DataSource=myiSeries; Trace=TraceJob, StartDebug, PrintJoblog;";
```

Important: If you use the StartDebug trace option, you must ensure your job log is saved; otherwise, the debug information from the job log is lost when your job ends. To save the job log, include the PrintJoblog option when you specify Trace in your ConnectionString.

Table 4-3 describes the trace data that results when your connection ends.

Table 4-3 Output files resulting from using Trace property with your iDB2Connection

Trace option	Output file
DatabaseMonitor	File QUSRSYS/QNETxxxxxx, where xxxxxx is your QZDASOINIT job number.
StartDebug	Debug statements are placed into your QZDASOINIT job log.
PrintJoblog	The QZDASOINIT job log is spooled.
TraceJob	The job trace information is spooled.

iDB2Connection methods

In this section, we discuss some of the methods you can call on your iDB2Connection object.

BeginTransaction

BeginTransaction is used to start a transaction on the iSeries server. For more information about transactions and isolation levels, see “DefaultIsolationLevel” on page 59, “Transaction” on page 69, and 4.6.3, “Using transactions” on page 116.

There are two ways to call BeginTransaction. In the first example, BeginTransaction is called with no parameters. This will begin a new transaction, and the new transaction will run using the transaction isolation level specified by the connection’s *DefaultIsolationLevel* property, as shown in Example 4-37.

Example 4-37 BeginTransaction()

```
// Create and open a connection to the iSeries.  
iDB2Connection cn = new iDB2Connection("DataSource=myiSeries;");  
cn.Open();  
  
// Begin a transaction. The new transaction will inherit the  
// connection's DefaultIsolationLevel.  
iDB2Transaction txn = cn.BeginTransaction();
```

In Example 4-38, we call BeginTransaction and pass an IsolationLevel parameter.

Example 4-38 BeginTransaction(IsolationLevel)

```
// Create and open a connection to the iSeries.  
iDB2Connection cn = new iDB2Connection("DataSource=myiSeries;");  
cn.Open();  
  
// Begin a transaction. The new transaction will use the  
// isolation level ReadCommitted.  
iDB2Transaction txn = cn.BeginTransaction(System.Data.IsolationLevel.ReadCommitted);
```

ChangeDatabase

The ChangeDatabase method is part of the ADO.NET interface definition. It is meant for changing the database your application connects to. Because the iSeries database is an integral part of the operating system, there is no separate database to connect to, so you never need to call the ChangeDatabase method, and in most cases your application will receive an exception if you try to call it. It is provided only for compatibility reasons.

Close

After you have finished using your iDB2Connection object, you should always call Close(). Keeping a connection open when it is no longer needed wastes system resources and may prevent others from being able to connect. If a transaction is still in progress when Close() is called, the transaction is rolled back.

The behavior of Close depends on whether the connection is pooled. By default, the provider always uses connection pooling. When you open a pooled connection, the provider takes an iDB2Connection object out of the pool for use. When you close the pooled connection, that connection is released back into the pool so it can be reused. If you do not close the connection, it will not be returned to the pool and cannot be reused.

When connection pooling is turned off (Pooling=false in the ConnectionString), Close() releases the host server job servicing that connection. If you do not close the connection, the host server job is not released by the provider.

Example 4-39 is an example of using the Close method to close a connection.

Example 4-39 Open() and Close() a connection

```
// Create and open a connection to the iSeries.
iDB2Connection cn = new iDB2Connection("DataSource=myiSeries;");
cn.Open();

// Do some work ...

// Close the connection.
cn.Close();
```

CreateCommand

The CreateCommand method is one way to create an iDB2Command object. The new command object is associated with this connection, and if a transaction was previously started on this connection, the new command object is automatically initialized to run under that transaction. See Example 4-40 on page 66 for an example of using CreateCommand().

Open

Before you can do any work on the iSeries, you must have an open connection. After you create your iDB2Connection object and initialize the ConnectionString property, call the Open() method to make the connection to the iSeries. Your connection must be open before you can begin a transaction or execute a command. See Example 4-39, for an example of how to open your connection.

4.5.3 iDB2Command properties and methods

When you want to run a SQL statement, perform a query, or call a stored procedure on the iSeries, use a *command* object. With IBM.Data.DB2.iSeries, the command object is called iDB2Command. This object implements the System.Data.IDbCommand interface.

Creating an iDB2Command object

There are many different ways to create a command, including:

- Call iDB2Connection's CreateCommand() method as shown in Example 4-40.

Example 4-40 iDB2Connection.CreateCommand() method

```
iDB2Connection myconnection = new iDB2Connection("DataSource=myiSeries;");
iDB2Command cmd = myconnection.CreateCommand();
```

When you use the CreateCommand() method, the resulting iDB2Command object is automatically associated with the iDB2Connection. (In this example, the cmd object is associated with myconnection.) This method of creating a command object is particularly useful when your connection is in transaction mode, because the new command automatically becomes part of the transaction.

- Use the *new* operator to create the object. The IBM.Data.DB2.iSeries .NET provider includes many different ways to construct a new command object. Example 4-41 shows some variations.

Example 4-41 iDB2Command, new operator

```
static void Main(string[] args)
{
    // Create a connection, open it, and start a transaction
    iDB2Connection cn = new iDB2Connection("DataSource=myiSeries;");
    cn.Open();
    iDB2Transaction txn = cn.BeginTransaction(System.Data.IsolationLevel.ReadCommitted);

    // This is the CommandText we will use for our example
    String cmdtext = "select * from sampled.employee";

    // Create an empty command. The command is not associated with a connection.
    // To use this command, you need to set the CommandText, Connection,
    // and Transaction properties (if your connection is in transaction mode).
    iDB2Command cmd1 = new iDB2Command();

    // Create a command and include the command text.
    // The command is not associated with a connection.
    // To use this command, you need to set the Connection and
    // Transaction properties (if your connection is in transaction mode).
    iDB2Command cmd2 = new iDB2Command(cmdtext);

    // Create a command and include the command text.
    // The command is associated with connection 'cn'.
    // To use this command, you need to set the Transaction property
    // (if your connection is in transaction mode).
    iDB2Command cmd3 = new iDB2Command(cmdtext, cn);

    // Create a command and include the command text.
    // The command is associated with connection 'cn'.
    // The command type is initialized to indicate a stored procedure call.
    // To use this command, you need to set the Transaction property
    // (if your connection is in transaction mode)
    iDB2Command cmd4 = new iDB2Command("mysp", System.Data.CommandType.StoredProcedure, cn);

    // Close the connection.
    cn.Close();
}
```

iDB2Command properties

The iDB2Command object has several properties, some of which we showed in previous examples.

CommandText

The CommandText property contains the SQL statement you want to run on the iSeries. CommandText can be almost any valid SQL statement, including a query (SELECT statement), a stored procedure call, or a statement that does not return any results, such as an INSERT statement. See 4.2.4, “Unsupported features” on page 38 for a list of statements that are not supported. The CommandText can include parameter markers, which enable you to provide variable data each time you execute the command. Parameter markers are specified in the CommandText with the @ sign followed by the parameter name, such as @param-name. We talk about parameter markers more in 4.5.4, “Using parameters in your SQL statements” on page 74. For information about calling stored procedures with the IBM.Data.DB2.iSeries provider, see 4.5.5, “Calling stored procedures” on page 79. Example 4-42 shows how to use the CommandText property.

Example 4-42 Setting the CommandText property

```
// Create a new command.
// Set the CommandText to a SELECT statement.
iDB2Connection cn = new iDB2Connection("DataSource=myiSeries;");
iDB2Command cmd = cn.CreateCommand();
cmd.CommandText = "select * from sampledб.employee";
```

CommandTimeout

The CommandTimeout property can be used to set a limit on the amount of time (in seconds) the iSeries server may spend processing the command. CommandTimeout on the iSeries is not a true timeout; rather, it is used by the iSeries to estimate whether to attempt to execute the command. If the iSeries server determines that the command may take longer than the CommandTimeout, it will not attempt the command. In this case, an iDB2SQLException will be thrown, indicating a timeout condition. The default CommandTimeout value is 30 (seconds), but you can specify a value of 0 to indicate no timeout. Example 4-43 shows how to use the CommandTimeout property.

Example 4-43 Setting the CommandTimeout property

```
// Create a new command and associate it with our connection.
// Set the CommandTimeout to "no timeout".
iDB2Connection cn = new iDB2Connection("DataSource=myiSeries;");
iDB2Command cmd = new iDB2Command("select * from sampledб.employee", cn);
cmd.CommandTimeout = 0;
```

CommandType

The CommandType property is a System.Data.CommandType enumeration value. The CommandType determines how the provider interprets the CommandText. The default CommandType is Text, indicating a regular SQL statement. Use the default in most cases.

- ▶ **CommandType.Text:** With CommandType.Text, the CommandText contains the actual statement you want to execute on the iSeries. The provider does not have to manipulate the CommandText (much) before sending it to the iSeries. Example 4-44 illustrates the use of the CommandType.Text.

Example 4-44 Setting CommandType=Text

```
// Create a command, set the command text, and use
// CommandType.Text.
```

```
iDB2Command cmd = new iDB2Command();
cmd.CommandText = "select firstnme, lastname from employee";
cmd.CommandType = System.Data.CommandType.Text;
```

- ▶ **CommandType.StoredProcedure:** With **CommandType.StoredProcedure**, the **CommandText** contains a stored procedure name. Using **CommandType.StoredProcedure** is not as efficient as using **CommandType.Text**, even when you want to call a stored procedure. We talk more about calling stored procedures in 4.5.5, “Calling stored procedures” on page 79. Example 4-45 illustrates the use of **CommandType.StoredProcedure**:

Example 4-45 Setting CommandType=StoredProcedure

```
// Create a command, set the command text, and use
// CommandType.StoredProcedure
iDB2Command cmd = new iDB2Command();
cmd.CommandText = "sampledb.empinfo";
cmd.CommandType = System.Data.CommandType.StoredProcedure;
```

- ▶ **CommandType.TableDirect:** With **CommandType.TableDirect**, the **CommandText** contains a list of table names separated by commas. The provider simply adds **SELECT *** in front of the list, resulting in a query that returns a join of all the specified tables. You can accomplish the same effect by using **CommandType.Text** and setting your **CommandText** to the **SELECT** statement:

"SELECT * FROM TABLE1, SAMPLEDB.EMPLOYEE" or "SELECT * FROM EMPLOYEE"

Example 4-46 illustrates the use of **CommandType.TableDirect**.

Example 4-46 Setting CommandType=TableDirect

```
// Create a command, set the command text, and use
// CommandType.TableDirect
iDB2Command cmd = new iDB2Command();
cmd.CommandText = "sampledb.employee";
cmd.CommandType = System.Data.CommandType.TableDirect;

// You can also specify a list of tables, like this:
cmd.CommandText = "sampledb.employee, anotherTable, aThirdTable";
```

Connection

Before you can execute any command, the command must be associated with an **iDB2Connection** object. There are three ways to do this. The second method is the most efficient because it sets the **Connection** property and the **CommandText** at the same time.

- ▶ Create the command using the **iDB2Connection.CreateCommand()** method, for example:

```
iDB2Command cmd = myconnection.CreateCommand();
```

- ▶ Specify the connection when you construct the command, for example:

```
iDB2Command cmd = new iDB2Command("mySQLcommandText", myconnection);
```

- ▶ Set the **iDB2Command**'s **Connection** property, as shown in Example 4-47.

Example 4-47 Setting the Connection property

```
iDB2Connection cn = new iDB2Connection("DataSource=myiSeries;");
iDB2Command cmd = new iDB2Command();
cmd.Connection = cn;
```

Parameters

Each `iDB2Command` object has a collection of `iDB2Parameters` associated with it. The `Parameters` property is used to reference this parameter collection. This property is read-only. The parameter collection may be empty. Parameters can be added to or deleted from the command by using the `Parameters.Add` method, or by using the command's `DeriveParameters()` method. We discuss `Parameters` more in 4.5.4, "Using parameters in your SQL statements" on page 74.

Transaction

Transactions are used to ensure that groups of database operations are performed with integrity. Operations that are grouped in a transaction can be committed permanently to the database or rolled back in case of failure. We discuss transactions in more detail in 4.6.3, "Using transactions" on page 116.

With `IBM.Data.DB2.iSeries`, transactions affect all the commands that run under the connection that started the transaction. Because of this, you must ensure that any `iDB2Command` object that runs under a connection uses the same `iDB2Transaction` object as the `iDB2Connection`. There are two ways to set the `Transaction` property of an `iDB2Command` object:

- Create the command using a connection that has begun a transaction (Example 4-48).

Example 4-48 Setting the `Transaction` property using `CreateCommand()`

```
// Create a connection, open it, and begin a transaction.
iDB2Connection cn = new iDB2Connection("DataSource=myiSeries;");
cn.Open();
iDB2Transaction txn = cn.BeginTransaction(System.Data.IsolationLevel.ReadCommitted);

// Create a new command using this connection.
// The command will be associated with this connection,
// and the command's Transaction property is automatically
// set to the 'txn' transaction.
iDB2Command cmd = cn.CreateCommand();
```

- Manually set the `Transaction` property (Example 4-49).

Example 4-49 Setting the `transaction` property manually

```
// Create a new command.
// Manually associate the command with our connection,
// and manually set the command's Transaction property.
iDB2Command cmd = new iDB2Command();
cmd.Connection = cn;
cmd.Transaction = txn;
```

UpdatedRowSource

The `UpdatedRowSource` property is used only when your command is used with an `iDB2DataAdapter`. It tells the `iDB2DataAdapter` how to apply the result data and output parameters of the command to the `DataRow` when `iDB2DataAdapter.Update` is called. For more information, read about `IDbCommand.UpdatedRowSource` property in the MSDN Library Web site:

<http://msdn.microsoft.com/library/>

Select **.NET Development** → **.NET Framework SDK** → **.NET Framework** → **Reference** → **Class Library** → **System.Data** → **IDbCommand Interface** → **Properties** → **UpdatedRowSource Property**.

iDB2Command methods

In this section, we discuss some of the methods you can call on your iDB2Command object. Most of these methods are covered in more detail in other examples in this chapter.

Cancel

For the IBM.Data.DB2.iSeries provider, the Cancel method is a no-op. It is included for compatibility purposes only because the IDbCommand interface defines a Cancel method. Calling Cancel() in effect does nothing.

DeriveParameters

Many of the commands you execute through the provider use parameters (see 4.5.4, “Using parameters in your SQL statements” on page 74). Before executing the command, the provider must map the parameter data you provide to the parameter value on the iSeries. To make this job easier, use the DeriveParameters() method. DeriveParameters is an extremely useful feature of the IBM.Data.DB2.iSeries provider. Some providers require you to create a CommandBuilder in order to derive parameter information (and then they only derive parameter information for stored procedures), but the IBM.Data.DB2.iSeries provider can derive parameter information for *any* command. In fact, we recommend using DeriveParameters() instead of defining parameters yourself. Because the provider always obtains a statement’s parameter information before executing a command, you do not save a trip to the host by defining your own parameters.

Example 4-50 illustrates the use of the DeriveParameters() method. This example uses the STAFF table in the SAMPLEDB schema. See 1.4, “DB2 UDB for iSeries sample schema” on page 8 for information about setting up the SAMPLEDB schema.

Example 4-50 Using the DeriveParameters method

```
// Create and open a connection to the iSeries.
iDB2Connection cn = new iDB2Connection("DataSource=myiSeries;DefaultCollection=sampled;");
cn.Open();

// Create a command object with parameter markers.
String s = "insert into staff values(@id, @name, @dept, @job, @years, @salary, @comm)";
iDB2Command cmd = new iDB2Command(s, cn);

// Derive the parameter information
cmd.DeriveParameters();

// Display the parameter descriptions
for (int i=0; i<cmd.Parameters.Count; i++)
{
    iDB2Parameter p = cmd.Parameters[i];
    Console.WriteLine("Parameter name: " + p.ParameterName);
    Console.WriteLine("  type:      " + p.iDB2DbType.ToString());
    Console.WriteLine("  direction: " + p.Direction.ToString());

    switch(p.iDB2DbType)
    {
        case iDB2DbType.iDB2Decimal:
        case iDB2DbType.iDB2Numeric:
            Console.WriteLine("    precision: " + p.Precision.ToString());
            Console.WriteLine("    scale:     " + p.Scale.ToString());
            break;

        case iDB2DbType.iDB2Binary:
        case iDB2DbType.iDB2Char:
        case iDB2DbType.iDB2CharBitData:
```

```

        case iDB2DbType.iDB2Graphic:
        case iDB2DbType.iDB2VarBinary:
        case iDB2DbType.iDB2VarChar:
        case iDB2DbType.iDB2VarCharBitData:
        case iDB2DbType.iDB2VarGraphic:
            Console.WriteLine("    size:        " + p.Size.ToString());
            break;

        default:
            break;
    }
}

// Dispose the command since we no longer need it.
cmd.Dispose();

// Close the connection.
cn.Close();

```

When you run this example, you can see that the provider knows the data type of each parameter and it knows that these are input parameters. The parameter name is the same name that was specified in the command text (in this example, @id, @name, @dept). We could call these parameters anything we want, such as @abxkas1, @abxkas2, and @abxkas3. We chose to make our parameter names match the column names from the table to make the code easier to read.

Dispose

When you are finished using any .NET object, it is a good idea to call its `Dispose()` method if one exists. `Dispose` cleans up resources associated with the object. For the `iDB2Command` object, `Dispose` cleans up the `iSeries` host resources used by the command (for example, any result data structure and request handle still in use). See Example 4-50 on page 70, for a sample that shows the `Dispose()` method.

ExecuteNonQuery, ExecuteReader, and ExecuteScalar

Most of the work you perform on the `iSeries` with the `IBM.Data.DB2.iSeries` .NET provider is done when you execute a command. There are three ways to execute a command. The method you choose will depend on whether the command returns result data, which is data returned in an `iDB2DataReader` object when you call `ExecuteReader` to execute a query (such as a `SELECT` statement), or execute a stored procedure that returns a result set. Read more about choosing a method in 4.5.6, “Choosing your execute method” on page 86.

Before calling an execute method, you must make the command ready. Follow these steps to execute your command:

1. Create the `iDB2Command` object (see “Creating an `iDB2Command` object” on page 66).
2. Set the `CommandText` property (see “`CommandText`” on page 67).
3. Set the `Connection` property (see “`Connection`” on page 68).
4. Set other command properties as needed (see 4.5.3, “`iDB2Command` properties and methods” on page 65).
5. Create or generate the parameter information if the command has parameters (see “`DeriveParameters`” on page 70).
6. Execute the command using one of the execute methods: `ExecuteNonQuery()`, `ExecuteReader()`, or `ExecuteScalar()`.

Next we show three coding examples that demonstrate these execute methods. These examples use the ACT table in the SAMPLEDB schema. See 1.4, “DB2 UDB for iSeries sample schema” on page 8 for information about setting up the SAMPLEDB schema.

- ▶ Example 4-51 inserts three rows of data into the ACT table.
- ▶ Example 4-52 reads all the data in the table using a DataReader.
- ▶ Example 4-53 on page 73 returns a count of the number of rows in the table.

Example 4-51 Using the ExecuteNonQuery method

```
// This example shows how you can use ExecuteNonQuery
// to run a command that does not return any result data.

// Create and open a connection to the iSeries.
iDB2Connection cn = new iDB2Connection("DataSource=myiSeries;DefaultCollection=sampled;");
cn.Open();

// Create a command object, and initialize its
// CommandText to an INSERT statement that has
// three parameters.
String cmdText = "insert into act values(@actno, @actkwd, @actdesc)";
iDB2Command cmd = new iDB2Command(cmdText, cn);

// Derive the parameter information
cmd.DeriveParameters();

// Execute the insert statement three times,
// using variable parameter data.
cmd.Parameters["@actno"].Value = 190;
cmd.Parameters["@actkwd"].Value = "DESIGN";
cmd.Parameters["@actdesc"].Value = "DESIGN PROGRAMS";
cmd.ExecuteNonQuery();

cmd.Parameters["@actno"].Value = 200;
cmd.Parameters["@actkwd"].Value = "ARCH";
cmd.Parameters["@actdesc"].Value = "ARCHITECT PRODUCT";
cmd.ExecuteNonQuery();

cmd.Parameters["@actno"].Value = 210;
cmd.Parameters["@actkwd"].Value = "PLAY";
cmd.Parameters["@actdesc"].Value = "PLAN FUN STUFF";
cmd.ExecuteNonQuery();

// Dispose the command since we no longer need it.
cmd.Dispose();

// Close the connection.
cn.Close();
```

Example 4-52 reads all the data in the table using a DataReader.

Example 4-52 Using the ExecuteReader method

```
// This example shows how you can use ExecuteReader
// to run a command that returns result data.

// Create and open a connection to the iSeries.
iDB2Connection cn = new iDB2Connection("DataSource=myiSeries;DefaultCollection=sampled;");
cn.Open();

// Create a command object, and initialize its
```

```

// CommandText to a SELECT statement.
iDB2Command cmd = new iDB2Command("select * from act", cn);

// Execute the SELECT statement. Because this command returns
// query result data, we use ExecuteReader() to read the
// data into an iDB2DataReader.
iDB2DataReader dr = cmd.ExecuteReader();

// Now, read the data back and print the results.
Console.WriteLine("ACTNO  ACTKWD  ACTDESC");
Console.WriteLine("-----  -----  -----");
while (dr.Read() == true)
{
    Console.WriteLine("{0}    {1}    {2}",
        dr.GetInt16(0).ToString("d3"),
        dr.GetString(1),
        dr.GetString(2));
}

// Close the DataReader since we are done reading the data.
dr.Close();

// Dispose the command since we no longer need it.
cmd.Dispose();

// Close the connection.
cn.Close();

```

Example 4-53 returns a count of the number of rows in the table.

Example 4-53 Using the ExecuteScalar method

```

// This example shows how you can use ExecuteScalar
// to run a command that returns the first column of
// the first row of a query.

// Create and open a connection to the iSeries.
iDB2Connection cn = new iDB2Connection("DataSource=myiSeries;DefaultCollection=sampledb;");
cn.Open();

// Create a command object, and initialize its
// CommandText to a SELECT COUNT(*) statement.
// This returns the number of rows in the table.
iDB2Command cmd = new iDB2Command("select count(*) from act", cn);

// Execute the SELECT statement. Because this command returns
// only a single value, we use ExecuteScalar() to read the
// result. While we could use ExecuteReader() instead,
// ExecuteScalar is more efficient, because there is less
// overhead.
int rowcount = (int)cmd.ExecuteScalar();
Console.WriteLine("There are {0} rows in the table.", rowcount);

// Dispose the command since we no longer need it.
cmd.Dispose();

// Close the connection.
cn.Close();

```

Prepare

The Prepare() method is used by the provider to prepare the statement for execution. With IBM.Data.DB2.iSeries, your application does not have to call Prepare directly. The provider always prepares statements on your behalf before executing them. It also prepares the statement when you call DeriveParameters(). Prepare can be used if you want to verify that the command is valid before calling one of the iDB2Command's Execute methods. A command that is not valid will throw an exception when Prepare() is called. For example, this could happen if you prepare a statement that issues a SELECT from a table, and that table does not exist. Example 4-54 illustrates an application that uses Prepare() to show that a valid statement prepares successfully and an invalid statement generates an error. It uses the EMPLOYEE table in the SAMPLEDB schema. See 1.4, "DB2 UDB for iSeries sample schema" on page 8 for information about setting up the SAMPLEDB schema.

Example 4-54 Using the Prepare method

```
// Create and open a connection to the iSeries.
iDB2Connection cn = new iDB2Connection("DataSource=myiSeries;DefaultCollection=sampled;");
cn.Open();

// Create a command object, and initialize its
// CommandText to something that is valid.
iDB2Command cmd1 = new iDB2Command("select * from employee", cn);

// Create a command object, and initialize its
// CommandText to something that is invalid.
iDB2Command cmd2 = new iDB2Command("select * from oops", cn);

// Prepare the commands to see if they are valid.
try
{
    cmd1.Prepare();
    Console.WriteLine("Command1 (" + cmd1.CommandText + ") is valid.");
    cmd2.Prepare();
    Console.WriteLine("Command2 (" + cmd2.CommandText + ") is valid.");
}
catch (Exception e)
{
    Console.WriteLine("The command was not prepared because an exception occurred.");
    Console.WriteLine("Exception: " + e.Message);
}

// Dispose the commands since we no longer need them.
cmd1.Dispose();
cmd2.Dispose();

// Close the connection.
cn.Close();
```

4.5.4 Using parameters in your SQL statements

Many of our previous examples use simple SQL statements without parameters. In this section, we discuss how to use SQL statements that contain parameters. We use the CL_SCHED table in the SAMPLEDB schema. See 1.4, "DB2 UDB for iSeries sample schema" on page 8 for information about setting up the SAMPLEDB schema.

Why you should use parameters in your SQL statements

When sending SQL statements to the iSeries to perform work on your behalf, it is often helpful to use variable parameters. Parameters can improve your performance and make your application more flexible. For example, say you want to execute the following three statements to insert data into the CL_SCHED table:

```
INSERT INTO CL_SCHED VALUES('043:LLD', 7, '08:00:00', '10:00:00')
INSERT INTO CL_SCHED VALUES('047:DRD', 4, '07:45:00', '08:30:00')
INSERT INTO CL_SCHED VALUES('008:AMD', 5, '09:10:00', '10:40:00')
```

To accomplish this, you could write the code shown in Example 4-55.

Example 4-55 Calling INSERT using literal values

```
// Create a connection and open it.
iDB2Connection cn = new iDB2Connection("DataSource=myiSeries;DefaultCollection=sampledb;");
cn.Open();

// Insert three records into the table.
iDB2Command cmd = new iDB2Command();
cmd.Connection = cn;
cmd.CommandText = "INSERT INTO CL_SCHED VALUES('043:LLD', 7, '08:00:00', '10:00:00')";
cmd.ExecuteNonQuery();
cmd.CommandText = "INSERT INTO CL_SCHED VALUES('047:DRD', 4, '07:45:00', '08:30:00')";
cmd.ExecuteNonQuery();
cmd.CommandText = "INSERT INTO CL_SCHED VALUES('008:AMD', 5, '09:10:00', '10:40:00')";
cmd.ExecuteNonQuery();

// Dispose the command since we no longer need it.
cmd.Dispose();

// Close the connection.
cn.Close();
```

While this approach works, it does not give you the most efficient performance on the iSeries. Instead, a more efficient approach uses parameters, as in Example 4-56.

Example 4-56 Calling INSERT using parameters

```
// Create a connection and open it.
iDB2Connection cn = new iDB2Connection("DataSource=myiSeries;DefaultCollection=sampledb;");
cn.Open();

// Create a command using parameter markers, and
// let the iSeries derive the parameter information.
iDB2Command cmd = cn.CreateCommand();
cmd.CommandText = "insert into cl_sched values(@code, @day, @start, @end)";
cmd.DeriveParameters();

// Insert three records into the table using parameters.
cmd.Parameters["@code"].Value = "043:LLD";
cmd.Parameters["@day"].Value = 7;
cmd.Parameters["@start"].Value = "08:00:00";
cmd.Parameters["@end"].Value = "10:00:00";
cmd.ExecuteNonQuery();

cmd.Parameters["@code"].Value = "047:DRD";
cmd.Parameters["@day"].Value = 4;
cmd.Parameters["@start"].Value = "07:45:00";
cmd.Parameters["@end"].Value = "08:30:00";
```

```

cmd.ExecuteNonQuery();

cmd.Parameters["@code"].Value = "008:AMD";
cmd.Parameters["@day"].Value = 5;
cmd.Parameters["@start"].Value = "09:10:00";
cmd.Parameters["@end"].Value = "10:40:00";
cmd.ExecuteNonQuery();

// Dispose the command since we no longer need it.
cmd.Dispose();

// Close the connection.
cn.Close();

```

Although the second example (Example 4-56 on page 75) looks like more work, it is the preferred method.

Every time you execute a new statement on the iSeries, the iSeries SQL optimizer must do extra work to build a plan for executing the statement. Therefore, executing the three INSERT statements in Example 4-55 on page 75 causes the optimizer to build plans for three different statements. By comparison, if you use a single statement and simply change the parameter values each time, as in Example 4-56 on page 75, the optimizer only has to generate the statement information the first time. Using parameters in this way can help your performance when the statement is executed many times.

Using parameters in your SQL statements also gives you more flexibility. For instance, you could write an application that takes as input a number corresponding to the day of the week (1-7), and select all of the classes in the CL_SCHED table that occur on that day. Your parameterized statement might look like this:

```
SELECT * FROM CL_SCHED WHERE (DAY = @day)
```

How to use parameters in your SQL statements

This section is about using parameters in SQL statements. As in the previous examples, we insert values into the CL_SCHED table in the SAMPLEDB schema. See 1.4, “DB2 UDB for iSeries sample schema” on page 8 for information about setting up the SAMPLEDB schema. Table 4-4 describes column names and data types in the CL_SCHED table.

Table 4-4 Column names and data types in the CL_SCHED table

Column name	Column data type
CLASS_CODE	CHAR(7)
DAY	SMALLINT
STARTING	TIME
ENDING	TIME

To use parameters, follow these steps:

1. Include *parameter markers* in your CommandText. A parameter marker tells the provider that you will use a parameter to fill in the specified value. With IBM.Data.DB2.iSeries, parameter markers are names prefixed with the @ character. In Example 4-57 on page 77, @code, @day, @start, and @end are parameter markers.

Example 4-57 CommandText that includes parameter markers

```
cmd.CommandText = "insert into sampledб.cl_sched values(@code, @day, @start, @end)";
```

2. Define an iDB2Parameter object to go with each parameter marker. The easiest way to create iDB2Parameter objects is to use the command's DeriveParameters() method, as shown in Example 4-58. DeriveParameters() gets the parameter information from the iSeries host and creates an iDB2Parameter for each parameter marker in the CommandText. It initializes the parameter information and adds the parameters to the command's Parameters collection.

Example 4-58 Adding parameters to your command using DeriveParameters()

```
// Create a command object that uses parameter markers.
String cmdText = "insert into sampledб.cl_sched values(@code, @day, @start, @end)";
iDB2Command cmd = new iDB2Command(cmdText, cn);

// Derive the parameter information.
cmd.DeriveParameters();
```

If you instead want to create your own parameter definitions, you can manually create the parameters and add them to the collection as shown in Example 4-59.

Example 4-59 Adding parameters to your command using Parameters.Add

```
// Define the parameters.
cmd.Parameters.Add("@code", iDB2DbType.iDB2Char, 7);
cmd.Parameters.Add("@day", iDB2DbType.iDB2SmallInt);
cmd.Parameters.Add("@start", iDB2DbType.iDB2Date);
cmd.Parameters.Add("@end", iDB2DbType.iDB2Date)
```

Note: There are different ways to create your own parameters. The provider supports several versions of the Parameters.Add method and different ways to construct an iDB2Parameter object. This example shows only one way to create parameters.

3. After you have defined your parameters, you must set the parameter values before calling your Execute method, as shown in Example 4-60.

Example 4-60 Setting parameter values

```
// Set the parameter values
cmd.Parameters["@code"].Value = "043:LLD";
cmd.Parameters["@day"].Value = 7;
cmd.Parameters["@start"].Value = "08:00:00";
cmd.Parameters["@end"].Value = "10:00:00";
```

Tip: If you want to send a null parameter value, set the parameter value to DBNull; for example:

```
cmd.Parameters["@day"].Value = System.DBNull.Value;
```

4. Execute your command. When you call one of the Execute methods, the parameter values are sent to the iSeries host. Example 4-61 on page 78 puts all of the steps together to show how you can insert the variable data into your table.

Example 4-61 Executing the command

```
// Create a connection and open it.
iDB2Connection cn = new iDB2Connection("DataSource=myiSeries;");
cn.Open();

// Create a command object that uses parameter markers.
String cmdText = "insert into sampledб.cl_sched values(@code, @day, @start, @end)";
iDB2Command cmd = new iDB2Command(cmdText, cn);

// Define the parameters.
cmd.Parameters.Add("@code", iDB2DbType.iDB2Char, 7);
cmd.Parameters.Add("@day", iDB2DbType.iDB2SmallInt);
cmd.Parameters.Add("@start", iDB2DbType.iDB2Date);
cmd.Parameters.Add("@end", iDB2DbType.iDB2Date);

// Set the parameter values
cmd.Parameters["@code"].Value = "043:LLD";
cmd.Parameters["@day"].Value = 7;
cmd.Parameters["@start"].Value = "08:00:00";
cmd.Parameters["@end"].Value = "10:00:00";

// Execute the command.
cmd.ExecuteNonQuery();

// Dispose the command since we no longer need it.
cmd.Dispose();

// Close the connection.
cn.Close();
```

With the IBM.Data.DB2.iSeries provider, parameter markers in your CommandText are prefixed with the @ character. To refer to a parameter, use the parameter name (@code or @day in the previous example). We call this type of parameter a *named parameter*. Named parameters are easy to use because we can assign meaningful names to our parameters and refer to them by name instead of by their position in the Parameters collection; for example:

```
cmd.Parameters["@param-name"].Value = 1;
```

If you are familiar with OLE DB or ODBC, you may recall that in those interfaces, parameter markers are indicated by using a question mark; for example:

```
insert into cl_sched values(?, ?, ?, ?)
```

We call this type of parameter an *unnamed parameter*. Although you can use the ? character to mark your parameters with the IBM.Data.DB2.iSeries provider, this method is not as flexible as using named parameters. If you use the ? character to mark your parameters, then you can only refer to the parameter by its index in the collection; for example:

```
cmd.Parameters[1].Value = 1;
```

This usage can be problematic. For example, if you want to add more parameters before existing parameters, you must make sure to renumber all of the existing parameter indexes to refer to their new position within the Parameters collection. Also, if you have a large number of parameters to keep track of, it is easier to make a mistake and refer to the wrong parameter.

Note: You cannot mix named parameters and unnamed parameters in the same command. You must use one or the other.

4.5.5 Calling stored procedures

Many applications can benefit from using stored procedures. Stored procedures enable you to perform many operations on the iSeries by using a single CALL statement. They also give you a way to call a program on the iSeries written in a supported programming language, even if the program does not contain any SQL statements. For more information about stored procedures, go to the iSeries Information Center at:

<http://www.iseries.ibm.com/infocenter>

Select **Database** → **Programming** → **SQL programming** → **Routines** → **Stored Procedures**.

Another good reference is the IBM Redbook *Stored Procedures, Triggers and User Defined Functions on DB2 Universal Database for iSeries*, SG24-6503. It includes a wealth of information about stored procedures on the iSeries.

With the IBM.Data.DB2.iSeries .NET provider, you can call a stored procedure using either of two methods:

- ▶ Use `CommandType.Text`, and set the `CommandText` to include the CALL statement, the stored procedure name, and the parameter markers.
- ▶ Use `CommandType.StoredProcedure`, and set the `CommandText` to the stored procedure name only.

The first method is preferred. It gives better performance overall and requires less work to generate parameter information.

In this section, we show how to call a stored procedure using both methods. First, we create a stored procedure called EMPINFO that pulls records from the EMPLOYEE table in the SAMPLEDB schema. See 1.4, "DB2 UDB for iSeries sample schema" on page 8 for information about setting up the SAMPLEDB schema. We also show examples of how to call this stored procedure using both `CommandType.Text` and `CommandType.StoredProcedure`. Our examples use input, output, and return value parameter types.

Before we begin, run the code in Example 4-62 to create the EMPINFO stored procedure.

Example 4-62 Creating the EMPINFO stored procedure

```
// Create and open a connection to the iSeries.
iDB2Connection cn = new iDB2Connection("DataSource=myiSeries;");
cn.Open();

// Create a stored procedure.
// In: employee number
// Out: first name, last name, salary
// Return value: recommended salary increase
iDB2Command cmd = cn.CreateCommand();
cmd.CommandText = "create procedure sampledb.empinfo (" +
    "in empnum char(6), out fname varchar(12), " +
    "out lname varchar(15), out current decimal(9, 2)) " +
    "language sql " +
    "begin " +
    "    select firstnme, lastname, salary into " +
    "        fname, lname, current from sampledb.employee " +
    "        where empno = empnum;" +
    "    return 5000;" +
    "end";
cmd.ExecuteNonQuery();
```

```
// Dispose the command since we no longer need it.
cmd.Dispose();

// Close the connection.
cn.Close();
```

Calling stored procedures with CommandType.Text

The easiest way to call a stored procedure with IBM.Data.DB2.iSeries is to use `CommandType.Text`, and include the `CALL` statement, the stored procedure name, and the parameter markers in your `CommandText`. Using this method has several advantages, especially when you want to use `DeriveParameters()` to have the provider automatically generate the parameter information for you.

To call a stored procedure using `CommandType.Text`, follow these steps:

1. Set your `CommandText` to `CALL` followed by the stored procedure name and parameters, if the procedure takes parameters. If the procedure returns a return value, indicate that by using a return value parameter. Here are some sample `CommandText` values:

```
cmd.CommandText = "call sampledb.empinfo('000010', @fname, @lname, @salary)";
cmd.CommandText = "@rc = call empinfo(@empnum, @fname, @lname, @salary)";
cmd.CommandText = "call myschema.noparms()";
```

2. Set your `CommandType` to `CommandType.Text` (note, this is the default `CommandType`):

```
cmd.CommandType = System.Data.CommandType.Text;
```

3. Define your parameters (the preferred method uses `DeriveParameters` but you can also add them manually; see Example 4-59 on page 77):

```
cmd.DeriveParameters();
```

4. Set your input and input/output parameter values, for example:

```
cmd.Parameters["@empnum"].Value = "000010";
```

5. Call your stored procedure using one of the `Execute` methods, for example:

```
cmd.ExecuteNonQuery();
```

Example 4-63 that calls the `EMPINFO` stored procedure we created earlier.

Example 4-63 Calling the EMPINFO stored procedure using CommandType.Text

```
// Create and open a connection to the iSeries.
iDB2Connection cn = new iDB2Connection("DataSource=myiSeries;");
cn.Open();

// Setup the command to call the stored procedure EMPINFO.
iDB2Command cmd = cn.CreateCommand();
cmd.CommandText = "@rc = call sampledb.empinfo(@employeeId, @firstname, @lastname,
@current_salary)";

// Let the provider generate the parameter information
cmd.DeriveParameters();

// Set the input parameter to one of the employees in the
// SAMPLEDB.EMPLOYEE table. For this example, we hard-code
// the employee id.
cmd.Parameters["@employeeId"].Value = "000180";

// Execute the command. Because the command does not return
// a result set, we use ExecuteNonQuery.
cmd.ExecuteNonQuery();
```

```
// Now, print out the employee first name, last name,
// and current salary.
Console.WriteLine("{0} {1}'s current salary: {2}",
    cmd.Parameters["@firstname"].Value,
    cmd.Parameters["@lastname"].Value,
    cmd.Parameters["@current_salary"].Value.ToString());

// The return value from our sample stored procedure is the
// recommended salary increase. Display that now.
Console.WriteLine("Recommended salary increase: " +
    cmd.Parameters["@rc"].Value.ToString());

// Dispose the command since we no longer need it.
cmd.Dispose();

// Close the connection.
cn.Close();
```

In the previous example, you can see that we handle the input parameter (@employeeId), the output parameters (@firstname, @lastname, @current_salary), and the return value (@rc). Because we use DeriveParameters(), the provider determines the parameter types for us.

Calling stored procedures with CommandType.StoredProcedure

This section explains how to use CommandType.StoredProcedure. As we discuss earlier, when calling a stored procedure with the IBM.Data.DB2.iSeries provider, it is more efficient to use CommandType.Text (see “Calling stored procedures with CommandType.Text” on page 80). When you use CommandType.StoredProcedure, you should manually define the parameters using Parameters.Add instead of DeriveParameters(), and you should include both the stored procedure name and the schema name. Using DeriveParameters() with CommandType.StoredProcedure will cause at least one extra trip to the iSeries host (to determine the parameter count). If you do not qualify the stored procedure name with the schema name (for instance, if you use empinfo instead of sampledb.empinfo), then two additional trips to the iSeries host are made in order to determine the CURRENT_PATH setting, and to query the QSYS2/SYSPROCS view to see how many parameters exist for the stored procedure. You can save a lot of trouble by using CommandType.Text instead.

To call a stored procedure using CommandType.StoredProcedure, follow these steps:

1. Set your CommandText to the schema name and stored procedure name:

```
cmd.CommandText = "sampledb.empinfo";
```

2. Set your CommandType to CommandType.StoredProcedure:

```
cmd.CommandType = System.Data.CommandType.StoredProcedure;
```

3. Define your parameters manually, for example:

```
cmd.Parameters.Add("@employeeId", iDB2DbType.iDB2Char, 6);
cmd.Parameters["@employeeId"].Direction = System.Data.ParameterDirection.Input;
cmd.Parameters.Add("@firstname", iDB2DbType.iDB2VarChar, 12);
cmd.Parameters["@firstname"].Direction = System.Data.ParameterDirection.Output;
```

4. Set your input and input/output parameter values, for example:

```
cmd.Parameters["@employeeId"].Value = "000010";
```

5. Call your stored procedure using one of the Execute methods, for example:

```
cmd.ExecuteNonQuery();
```

Example 4-64 on page 82 calls the EMPINFO stored procedure that we created earlier.

Example 4-64 Calling the EMPINFO stored procedure using CommandType.StoredProcedure

```
// Create and open a connection to the iSeries.
iDB2Connection cn = new iDB2Connection("DataSource=myiSeries;");
cn.Open();

// Setup the command to call the stored procedure EMPINFO.
iDB2Command cmd = new iDB2Command("sampledb.empinfo", cn);
cmd.CommandType = System.Data.CommandType.StoredProcedure;

// Create the parameter definitions.
cmd.Parameters.Add("@rc", iDB2DbType.iDB2Integer);
cmd.Parameters["@rc"].Direction = System.Data.ParameterDirection.ReturnValue;

cmd.Parameters.Add("@employeeId", iDB2DbType.iDB2Char, 6);
cmd.Parameters["@employeeId"].Direction = System.Data.ParameterDirection.Input;

cmd.Parameters.Add("@firstname", iDB2DbType.iDB2VarChar, 12);
cmd.Parameters["@firstname"].Direction = System.Data.ParameterDirection.Output;

cmd.Parameters.Add("@lastname", iDB2DbType.iDB2VarChar, 15);
cmd.Parameters["@lastname"].Direction = System.Data.ParameterDirection.Output;

cmd.Parameters.Add("@current_salary", iDB2DbType.iDB2Decimal);
cmd.Parameters["@current_salary"].Precision = 9;
cmd.Parameters["@current_salary"].Scale = 2;
cmd.Parameters["@current_salary"].Direction = System.Data.ParameterDirection.Output;

// Set the input parameter to one of the employees in the
// SAMPLEDB.EMPLOYEE table. For this example, we hard-code
// the employee id.
cmd.Parameters["@employeeId"].Value = "000180";

// Execute the command. Because the command does not return
// a result set, we use ExecuteNonQuery.
cmd.ExecuteNonQuery();

// Now, print out the employee first name, last name,
// and current salary.
Console.WriteLine("{0} {1}'s current salary: {2}",
    cmd.Parameters["@firstname"].Value,
    cmd.Parameters["@lastname"].Value,
    cmd.Parameters["@current_salary"].Value.ToString());

// The return value from our sample stored procedure is the
// recommended salary increase. Display that now.
Console.WriteLine("Recommended salary increase: " +
    cmd.Parameters["@rc"].Value.ToString());

// Dispose the command since we no longer need it.
cmd.Dispose();

// Close the connection.
cn.Close();
```

We handle the input parameter (@employeeId), the output parameters (@firstname, @lastname, @current_salary), and the return value (@rc). Because we create the parameter definitions, more work is required. We must specify the parameter data type, direction (input, output, input/output, or return value), and size if it is a variable-length data type.

Note: When you add your own parameters with `CommandType.StoredProcedure`, you are not required to prefix the parameter names with the `@` character because you are not using parameter markers in your `CommandText`. We use the `@` here for consistency.

Using return value parameters

Example 4-63 on page 80 and Example 4-64 on page 82 showed how to use a return value parameter. If you define your own parameters, take special care with the return value parameter. The return value parameter (if you choose to use one) *must* be the first parameter in the `Parameters` collection. This means that you must either add it before you add the other parameters or use `cmd.Parameters.Insert` and insert the return value parameter at the first location (index value 0) in the `parameters` collection.

If you use `DeriveParameters()` to generate the parameter information, you must use `CommandType.Text` and specify the return value parameter in the call statement as in Example 4-63 on page 80; otherwise, no return value parameter is generated.

Calling stored procedures that return results

Example 4-63 on page 80 and Example 4-64 on page 82 call a stored procedure that uses input, output, and return value parameters. Now we show how to call a stored procedure that returns a result set. As with any stored procedure call, you can use input, output, input/output, and return value parameters as needed.

Create a stored procedure called `JOBINFO` that pulls records from the `EMPLOYEE` table in the `SAMPLEDB` schema and returns them as result data. See 1.4, “DB2 UDB for iSeries sample schema” on page 8 to set up the `SAMPLEDB` schema. Our example uses an input parameter to select all employees in the `EMPLOYEE` table whose job matches the input parameter.

First, run the code shown in Example 4-65 to create the `JOBINFO` stored procedure.

Example 4-65 Creating the JOBINFO stored procedure

```
// Create and open a connection to the iSeries.
iDB2Connection cn = new iDB2Connection("DataSource=myiSeries;");
cn.Open();

// Create a stored procedure that selects records
// from the EMPLOYEE table.
iDB2Command cmd = cn.CreateCommand();
cmd.CommandText = "create procedure sampledb.jobinfo (" +
    "in jobdesc char(8)) result sets 1 language sql " +
    "begin " +
    "    declare c1 cursor for select * from " +
    "        sampledb.employee where job=jobdesc; " +
    "    open c1; " +
    "    set result sets cursor c1; " +
    "end";
cmd.ExecuteNonQuery();

// Dispose the command since we no longer need it.
cmd.Dispose();

// Close the connection.
cn.Close();
```

Next, call the stored procedure that we just created. Because this stored procedure returns result data, we use `ExecuteReader` to read the results, as shown in Example 4-66 on page 84.

Example 4-66 Calling a stored procedure that returns a result set

```
// Create and open a connection to the iSeries.
iDB2Connection cn = new iDB2Connection("DataSource=myiSeries;");
cn.Open();

// Set up the stored procedure call.
// Let the provider generate the parameter information.
iDB2Command cmd = new iDB2Command("call sampledb.jobinfo(@jobdesc)", cn);
cmd.DeriveParameters();

// Set the input parameter to "DESIGNER".
cmd.Parameters["@jobdesc"].Value = "DESIGNER";

// Call the stored procedure. Because the procedure returns
// result data, we use ExecuteReader.
iDB2DataReader dr = cmd.ExecuteReader();

// Now, read each record and display the employee information.
Console.WriteLine("EMPNO First Name Last Name DPT Date hired Salary");
Console.WriteLine("-----");

while (dr.Read())
{
    Console.WriteLine("{0} {1} {2} {3} {4} {5}",
        dr.GetDB2Char(0).Value, // EMPNO (first field)
        dr.GetDB2VarChar(1).Value.PadRight(12), // FIRSTNAME (second field)
        dr.GetDB2VarChar(3).Value.PadRight(15), // LASTNAME (fourth field)
        dr.GetDB2Char(4).Value, // WORKDEPT (fifth field)
        dr.GetDB2Date(6).ToNativeFormat(), // HIREDATE (seventh field)
        dr.GetDB2Decimal(11).ToString()); // SALARY (12th field)
}

// Close the DataReader since we're done using it.
dr.Close();

// Dispose the command since we no longer need it.
cmd.Dispose();

// Close the connection.
cn.Close();
```

Calling stored procedures that return multiple results

There are times when you want to return more than one result set from a stored procedure. This is easy to do using the `iDB2DataReader` with the `NextResult` method.

Create a stored procedure called `DEPTINFO` that pulls records from the `EMPLOYEE` and `DEPARTMENT` tables in the `SAMPLEDB` schema and returns them as two result sets. See 1.4, “DB2 UDB for iSeries sample schema” on page 8 to set up the `SAMPLEDB` schema. Our stored procedure uses an input parameter to select these result sets:

- ▶ `SELECT * FROM EMPLOYEE WHERE WORKDEPT=@dept`
- ▶ `SELECT * FROM DEPARTMENT WHERE DEPTNO=@dept`

The stored procedure takes a three-character department number as input and returns two result sets. The first result set includes all of the employees in the department, and the second result set includes all of the information about the department.

First, run the code in Example 4-67 on page 85 to create the `DEPTINFO` stored procedure.

Example 4-67 Creating the DEPTINFO stored procedure

```
// Create and open a connection to the iSeries.
iDB2Connection cn = new iDB2Connection("DataSource=myiSeries;");
cn.Open();

// Create a stored procedure that selects records
// from the EMPLOYEE table and the DEPARTMENT table.
iDB2Command cmd = cn.CreateCommand();
cmd.CommandText = "create procedure sampledb.deptinfo (" +
    "in dept char(3)) result sets 2 language sql " +
    "begin " +
    "    declare c1 cursor with return for select * from " +
    "        sampledb.employee where workdept=dept; " +
    "    declare c2 cursor with return for select * from " +
    "        sampledb.department where deptno=dept; " +
    "    open c1; open c2; " +
    "end";
cmd.ExecuteNonQuery();

// Dispose the command since we no longer need it.
cmd.Dispose();

// Close the connection.
cn.Close();
```

Now call the stored procedure that we just created. Because this stored procedure returns result data, we use `ExecuteReader` to read the results, as shown in Example 4-68.

Example 4-68 Calling a stored procedure that returns multiple results

```
// Create and open a connection to the iSeries.
iDB2Connection cn = new iDB2Connection("DataSource=myiSeries;");
cn.Open();

// Set up the stored procedure call.
// Let the provider generate the parameter information.
iDB2Command cmd = new iDB2Command("call sampledb.deptinfo(@dept)", cn);
cmd.DeriveParameters();

// Set the input parameter to department "A00".
cmd.Parameters["@dept"].Value = "A00";

// Call the stored procedure. Because the procedure returns
// result data, we use ExecuteReader.
iDB2DataReader dr = cmd.ExecuteReader();

// Now, read each record from the first result set,
// and display the employee information.
Console.WriteLine("First result set: Employee information.");
Console.WriteLine("");
Console.WriteLine("EMPNO    First Name    Last Name        DPT Date hired Salary");
Console.WriteLine("-----");

while (dr.Read())
{
    Console.WriteLine("{0} {1} {2} {3} {4} {5}",
        dr.GetDB2Char(0).Value,           // EMPNO (first field)
        dr.GetDB2VarChar(1).Value.PadRight(12), // FIRSTNAME (second field)
        dr.GetDB2VarChar(3).Value.PadRight(15), // LASTNAME (fourth field)
        dr.GetDB2Char(4).Value,           // WORKDEPT (fifth field)
```

```

        dr.GetDB2Date(6).ToNativeFormat(), // HIREDATE (seventh field)
        dr.GetDB2Decimal(11).ToString()); // SALARY (12th field)
    }

    // Now that we're done reading the employees in the
    // department, display the department information from the
    // second result set.
    if (dr.NextResult() == true)
    {
        Console.WriteLine("");
        Console.WriteLine("Second result set: Department information.");
        Console.WriteLine("DEPT  DEPTNAME                MGRNO  ADMRDEPT");
        Console.WriteLine("----  -----");

        if (dr.Read())
        {
            Console.WriteLine("{0}    {1} {2} {3}",
                dr.GetString(0),      // DEPTNO (first field)
                dr.GetString(1),      // DEPTNAME (second field)
                dr.GetString(2),      // MGRNO (third field)
                dr.GetString(3));     // ADMRDEPT (fourth field)
        }
    }

    // Close the DataReader since we're done using it.
    dr.Close();

    // Dispose the command since we no longer need it.
    cmd.Dispose();

    // Close the connection.
    cn.Close();

```

If your stored procedure returns more than two result sets, call the DataReader's NextResult method to continue to the next result set. When there are no more, NextResult() returns false.

4.5.6 Choosing your execute method

As with any ADO.NET provider, IBM.Data.DB2.iSeries provides three ways to execute SQL statements: ExecuteNonQuery, ExecuteReader, and ExecuteScalar. The method you choose is determined by answering these questions:

- ▶ Does the statement return any result data?
- ▶ If result data is returned, is it a single item (say, one column of one row) or multiple items?

Result data is data returned from a query, such as the result of executing a SELECT statement. Result data can also be returned from a stored procedure call. If your statement does not return result data, then use ExecuteNonQuery. ExecuteNonQuery has an interesting side effect: If the statement you execute is an INSERT, UPDATE, or DELETE statement, then ExecuteNonQuery returns the number of rows inserted, updated, or deleted; otherwise, it returns -1. Here are some examples of statements that are best used with ExecuteNonQuery:

```

UPDATE EMPLOYEE SET SALARY=SALARY+5000 WHERE EMPNO='200340'
INSERT INTO SAMPLEDB.DEPARTMENT VALUES(@DEPTNO, @DEPTNAME, @ADMRDEPT, @LOCATION)

```

Tip: If you want to call a stored procedure that either does not return result data or returns result data that you do not care to read, use ExecuteNonQuery, because it uses less overhead than the other execute methods.

If your statement *does* return result data, then you must choose between `ExecuteReader` and `ExecuteScalar`. Use `ExecuteScalar` when your result data consists of a single item. `ExecuteScalar` uses much less overhead than `ExecuteReader` because it simply returns an object representing the first column of the first row of your result data. After calling `ExecuteScalar`, you have the data you need and no further work is required. An example of when you might want to use `ExecuteScalar` is when you execute the following queries:

```
SELECT COUNT(*) FROM SAMPLEDB.EMPLOYEE
SELECT SUM(SALARY) FROM SAMPLEDB.EMPLOYEE
```

Because `ExecuteReader` returns an `iDB2DataReader` object, it requires more work on your part. You must *read* the result data, *retrieve* the results you want, and then *close* the `iDB2DataReader` object when you are done. `ExecuteReader` is the choice for most queries.

Use the flow chart in Figure 4-15 to help you choose your `Execute` method.

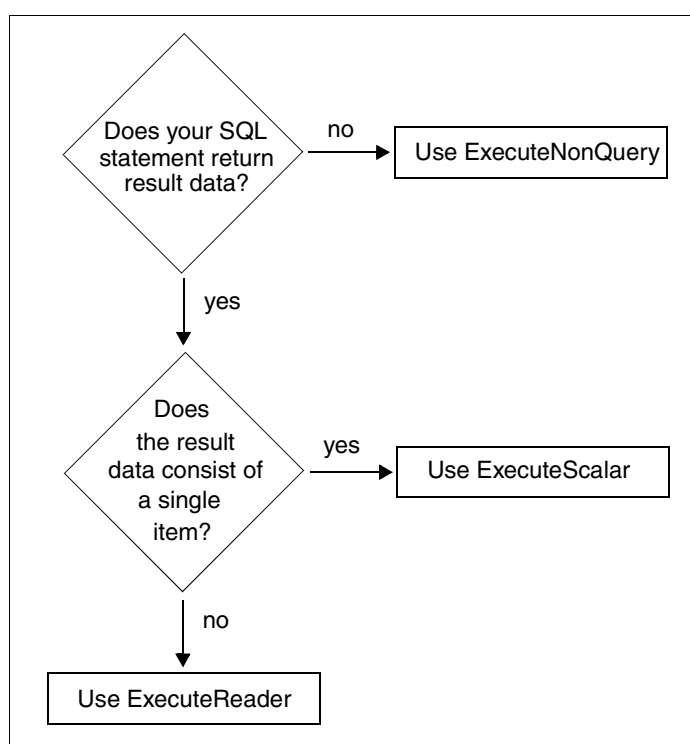


Figure 4-15 Choosing your `Execute` method

Sometimes when your command returns result data, you will choose to use an `iDB2DataAdapter` instead of using an `iDB2DataReader`. See 4.6.6, “Choosing between `iDB2DataReader` and `iDB2DataAdapter`” on page 127 for more information.

4.5.7 Provider data types

The `IBM.Data.DB2.iSeries` provider supports most data types that are supported by DB2 UDB for iSeries. Each of the supported data types has a corresponding provider data type. When performing operations with iSeries data, you may need to use these provider data types to take full advantage of the iSeries functionality. Table 4-5 on page 88 shows the supported iSeries data types and the provider and .NET framework types they correspond to.

Table 4-5 Data types supported by the IBM.Data.DB2.iSeries provider

iSeries data type	IBM.Data.DB2.iSeries provider data type name	Similar .NET framework data type
BIGINT	iDB2BigInt	Int64
BINARY	iDB2Binary	Byte[]
BLOB	iDB2Blob	Byte[]
CHAR	iDB2Char	String
CHAR with CCSID 65535 or CHAR FOR BIT DATA	iDB2CharBitData	Byte[]
CLOB	iDB2Clob	String
DATE	iDB2Date	DateTime
DBCLOB	iDB2DbClob	String
PACKED DECIMAL	iDB2Decimal	Decimal
DOUBLE	iDB2Double	Double
GRAPHIC	iDB2Graphic	String
INTEGER	iDB2Integer	Int32
ZONED DECIMAL	iDB2Numeric	Decimal
REAL or SINGLE	iDB2Real	Single
ROWID	iDB2Rowid	Byte[]
SMALLINT	iDB2SmallInt	Int16
TIME	iDB2Time	DateTime
TIMESTAMP	iDB2TimeStamp	DateTime
VARBINARY	iDB2VarBinary	Byte[]
VARCHAR	iDB2VarChar	String
VARCHAR with CCSID 65535 or VARCHAR FOR BIT DATA	iDB2VarCharBitData	Byte[]
VARGRAPHIC	iDB2VarGraphic	String

Note: The DATALINK data type is not supported by the IBM.Data.DB2.iSeries provider as of this writing. For more information, read 4.7.5, “Using DataLinks” on page 141.

The fixed-length string and binary types are always the same length, but variable-length types can vary in size. For example, if you read data from a CHAR(10) column that has only three characters of data, the provider returns the full 10 characters of data, including seven blanks at the end. If you do not want to see the blanks at the end, use the String .Trim() method to trim the blanks off. If you read data from a VARCHAR(10) column that has only three characters of data, the provider returns only the three characters of data.

Often, you can use provider data types interchangeably with the built-in .NET framework data types shown in Table 4-5. To illustrate this feature, we use the EMPLOYEE table in the SAMPLEDB schema. See 1.4, “DB2 UDB for iSeries sample schema” on page 8.

The EMPLOYEE table contains a CHAR(6) column called EMPNO. If you execute a query against the EMPLOYEE table using an iDB2DataReader, there are several ways to read the EMPNO column and get similar results. Example 4-69 shows how to read the CHAR(6) column into either a String or an iDB2Char variable and get the same results. It also shows that you can read EDLEVEL, a SMALLINT, into either an Int16 or an iDB2SmallInt variable and get the same results.

Example 4-69 Different ways to read a column using an iDB2DataReader

```
// Create and open a connection to the iSeries.
iDB2Connection cn = new iDB2Connection("DataSource=myiSeries;
DefaultCollection=sampldb;");
cn.Open();

// Create a command which will select records from the EMPLOYEE table
iDB2Command cmd = new iDB2Command("select * from employee", cn);

// Execute the command and get a DataReader we can use
// to read the data from the table.
iDB2DataReader dr = cmd.ExecuteReader();

if (dr.Read())
{
    // Read the first field, EMPNO, into a String variable.
    // Print the result.
    String empnoString = dr.GetString(0);
    Console.WriteLine("EMPNO as a String: " + empnoString);

    // Read the first field, EMPNO, into an iDB2Char variable.
    // Print the result.
    iDB2Char empnoiDB2Char = dr.GetiDB2Char(0);
    Console.WriteLine("EMPNO as an iDB2Char: " + empnoiDB2Char);

    // Read the ninth field, EDLEVEL, into an Int16 variable.
    // Print the result.
    Int16 edlevelInt16 = dr.GetInt16(8);
    Console.WriteLine("EDLEVEL as an Int16: " + edlevelInt16);

    // Read the ninth field, EDLEVEL, into an iDB2SmallInt variable.
    // Print the result.
    iDB2SmallInt edleveliDB2SmallInt = dr.GetiDB2SmallInt(8);
    Console.WriteLine("EDLEVEL as an iDB2SmallInt: " + edleveliDB2SmallInt);
}

// Close the DataReader since we no longer need it.
dr.Close();

// Dispose the command since we no longer need it.
cmd.Dispose();

// Close the connection.
cn.Close();
```

With some other data types, however, you must take precautions to ensure that the data is handled correctly. Next we discuss how to handle some data types that take special consideration: iDB2CharBitData, iDB2VarCharBitData, iDB2Date, iDB2Time, iDB2TimeStamp, iDB2Decimal, and iDB2Numeric.

iDB2CharBitData and iDB2VarCharBitData

The iDB2CharBitData and iDB2VarCharBitData data types represent character data (fixed and variable length) tagged on the host with the binary CCSID 65535. To tag a character column with CCSID 65535, specify FOR BIT DATA or CCSID 65535 when you define the column. In Example 4-70, we create a table called BITTABLE that contains both fixed and variable-length character columns, tagged with CCSID 65535. We also use both the FOR BIT DATA and CCSID 65535 methods for our example. We create an INSERT command that references each of the bit columns in the table, derive the command parameters, and display the parameter type to show how the provider maps parameter data types for these columns.

Example 4-70 Creating the BITTABLE table using char data tagged with CCSID 65535

```
// Create and open a connection to the iSeries.
iDB2Connection cn = new iDB2Connection("DataSource=myiSeries;DefaultCollection=sampledb;");
cn.Open();

// Create a command which will create a table
// containing two character columns tagged with
// CCSID 65535.
iDB2Command cmd = cn.CreateCommand();
cmd.CommandText = "create table bittable (" +
    "bit1 char(10) for bit data,      " +
    "bit2 char(10) ccsid 65535,      " +
    "varbit1 varchar(10) for bit data, " +
    "varbit2 varchar(10) ccsid 65535)";

// Execute the command.
cmd.ExecuteNonQuery();

// Now create an INSERT command, and derive the parameters.
cmd.CommandText = "insert into bittable values(@bit1, @bit2, @varbit1, @varbit2)";
cmd.DeriveParameters();

// Examine the parameter information. You can see that
// the fixed-length parameters are of type iDB2CharBitData,
// and the variable-length parameters are of type iDB2VarCharBitData.
Console.WriteLine(cmd.Parameters["@bit1"].iDB2DbType.ToString());
Console.WriteLine(cmd.Parameters["@bit2"].iDB2DbType.ToString());
Console.WriteLine(cmd.Parameters["@varbit1"].iDB2DbType.ToString());
Console.WriteLine(cmd.Parameters["@varbit2"].iDB2DbType.ToString());

// Dispose the command since we no longer need it.
cmd.Dispose();

// Close the connection.
cn.Close();
```

Char with CCSID 65535 was originally intended to be used with binary data. Over time, however, applications began to use this type of column to hold non-binary data, and they expected to be able to read and write this data using a non-binary CCSID. Because of this, using char with CCSID 65535 data can be problematic. Whenever possible, your application should use a more appropriate method to store your data. If the data is meant to be human-readable (for instance, EBCDIC or ASCII data), then use the CHAR or VARCHAR data type, and make sure that the data in the table is tagged with the appropriate CCSID. If the data is binary, then use the BINARY or VARBINARY data type, which was introduced with IBM DB2 UDB for iSeries in its V5R3M0 release.

As shown in Table 4-5 on page 88, the IBM.Data.DB2.iSeries provider treats bit data (char with CCSID 65535) as a binary array of bytes. This means that retrieving the Value property of an iDB2CharBitData or iDB2VarCharBitData object will return an array of bytes. Similarly, using a DataAdapter with this type of column results in binary data. If your application expects bit data to be translated to a non-binary value, you must take explicit action to make this translation happen. Run the code in Example 4-71 to insert some non-binary data into the BITTABLE table that was created in Example 4-70 on page 90. Although the bit columns are tagged with the binary CCSID 65535, the literal string values that we insert into the table are encoded using the server job's CCSID because they are character string literals.

Example 4-71 Inserting non-binary data into char columns tagged with CCSID 65535

```
// Create and open a connection to the iSeries.
iDB2Connection cn = new iDB2Connection("DataSource=myiSeries;DefaultCollection=sampldb;");
cn.Open();

// Create a command and use it to insert non-binary data
// into the bit columns.
iDB2Command cmd = cn.CreateCommand();
cmd.CommandText = "insert into bittable values('ABCDE', 'abcde', 'VWXYZ', 'vwxyz')";
cmd.ExecuteNonQuery();

// Dispose the command since we no longer need it.
cmd.Dispose();

// Close the connection.
cn.Close();
```

Reading non-binary data tagged with CCSID 65535

You can use one of the following options to read the non-binary data in the BITTABLE table:

- Alter the table to have the correct CCSID. (This is the recommended solution.) When you alter the table this way, the data is no longer treated as bit data, but as translatable character data. Read about the ALTER TABLE command in the DB2 Universal Database for iSeries SQL Reference, which you can find in the iSeries Information Center at:

<http://www.iseries.ibm.com/infocenter>

Select **Database** → **Reference** → **SQL Reference** → **Statements** → **ALTER TABLE**.

Example 4-72 shows how to alter the BITTABLE if your non-binary data is encoded in CCSID 37.

Example 4-72 Using ALTER TABLE to change the CCSID of a binary character column

```
// Create and open a connection to the iSeries.
iDB2Connection cn = new iDB2Connection("DataSource=myiseries;
DefaultCollection=sampldb;");
cn.Open();

// Create a command and use it to change the table
// definition for BITTABLE
iDB2Command cmd = cn.CreateCommand();
cmd.CommandText = "alter table bittable " +
    "alter column bit1 set data type char(10) ccsid 37 " +
    "alter column bit2 set data type char(10) ccsid 37 " +
    "alter column varbit1 set data type varchar(10) ccsid 37 " +
    "alter column varbit2 set data type varchar(10) ccsid 37";
cmd.ExecuteNonQuery();

// Dispose the command since we no longer need it.
```

```
cmd.Dispose();

// Close the connection.
cn.Close();
```

Note: If your data is encoded using a CCSID other than 37, use that CCSID in this example instead of 37.

- Alter your SQL SELECT statement to explicitly cast the data to the correct CCSID. When you cast the data to the correct CCSID, the data is no longer treated as bit data, but as translatable character data. Example 4-73 shows how to modify the SELECT statement to ensure that non-binary bit data is translated using CCSID 37.

Example 4-73 Using a CAST statement to select data marked with CCSID 65535

```
// Create and open a connection to the iSeries.
iDB2Connection cn = new iDB2Connection("DataSource=myiseries;DefaultCollection=sampledb;");
cn.Open();

// Create a command to query the BITTABLE
iDB2Command cmd = cn.CreateCommand();
cmd.CommandText = "select cast(bit1 as char(10) ccsid 037), " +
    "cast(bit2 as char(10) ccsid 037), " +
    "cast(varbit1 as varchar(10) ccsid 037), " +
    "cast(varbit2 as varchar(10) ccsid 037) from bittable";

// Run the command and get a DataReader in return.
iDB2DataReader dr = cmd.ExecuteReader();
if (dr.Read())
{
    // Display the field type for each field.
    // You can see that the data is now treated
    // as regular character data.
    Console.WriteLine(dr.GetDB2FieldType(0).ToString());
    Console.WriteLine(dr.GetDB2FieldType(1).ToString());
    Console.WriteLine(dr.GetDB2FieldType(2).ToString());
    Console.WriteLine(dr.GetDB2FieldType(3).ToString());
}

// Close the DataReader since we no longer need it.
dr.Close();

// Dispose the command since we no longer need it.
cmd.Dispose();

// Close the connection.
cn.Close();
```

Note: Altering the SELECT statement as shown in Example 4-73 enables the iDB2DataAdapter to treat bit data as translatable character data.

- Use a DataReader with GetString() to read the data. GetString() translates the binary data using the host server job's CCSID. Example 4-74 on page 93 shows how to use the DataReader with GetString to read the non-binary data.

Example 4-74 Using a DataReader with GetString to read non-binary data marked with CCSID 65535

```
// Create and open a connection to the iSeries.
iDB2Connection cn = new iDB2Connection("DataSource=myiseries;DefaultCollection=sampledb;");
cn.Open();

// Create a command to query the BITTABLE
iDB2Command cmd = new iDB2Command("select * from bittable", cn);

// Run the command and read from the DataReader.
iDB2DataReader dr = cmd.ExecuteReader();
if (dr.Read())
{
    // Read the data using GetString().
    Console.WriteLine("DataReader.GetString method: ");
    Console.WriteLine(" {0} {1} {2} {3}",
        dr.GetString(0), dr.GetString(1), dr.GetString(2), dr.GetString(3));
}

// Close the DataReader since we no longer need it.
dr.Close();

// Dispose the command since we no longer need it.
cmd.Dispose();

// Close the connection.
cn.Close();
```

- Use a DataReader to retrieve an iDB2CharBitData or iDB2VarCharBitData object, then use the ToString() or ToString(ccsid) method to read the data. ToString() translates the binary data using the host server job's CCSID, and ToString(ccsid) enables you to specify which CCSID to use for translation. Example 4-75 shows how to use the DataReader with both of these methods to read the non-binary data.

Example 4-75 Using a DataReader with GetiDB2CharBitData to read data marked with CCSID 65535

```
// Create and open a connection to the iSeries.
iDB2Connection cn = new iDB2Connection("DataSource=myiseries;DefaultCollection=sampledb;");
cn.Open();

// Create a command to query the BITTABLE
iDB2Command cmd = new iDB2Command("select * from bittable", cn);

// Run the command and read from the DataReader.
iDB2DataReader dr = cmd.ExecuteReader();
if (dr.Read())
{
    // Read the data and retrieve the values into
    // objects of the appropriate data type.
    iDB2CharBitData bit1 = dr.GetiDB2CharBitData(0);
    iDB2CharBitData bit2 = dr.GetiDB2CharBitData(1);
    iDB2VarCharBitData varbit1 = dr.GetiDB2VarCharBitData(2);
    iDB2VarCharBitData varbit2 = dr.GetiDB2VarCharBitData(3);

    // Now use ToString() to display the values.
    // Data is translated using the server job's CCSID.
    Console.WriteLine("ToString() method:");
    Console.WriteLine(" {0} {1} {2} {3}",
        bit1.ToString(), bit2.ToString(), varbit1.ToString(), varbit2.ToString());

    // You can also explicitly specify the CCSID
```

```
// using the example below.
Console.WriteLine("ToString(ccsid) method:");
Console.WriteLine(" {0} {1} {2} {3}",
    bit1.ToString(37), bit2.ToString(37), varbit1.ToString(37), varbit2.ToString(37));
}

// Close the DataReader since we no longer need it.
dr.Close();

// Dispose the command since we no longer need it.
cmd.Dispose();

// Close the connection.
cn.Close();
```

Note: If your data is encoded using a CCSID other than 37, use that CCSID in this example instead of 37.

iDB2Date, iDB2Time, and iDB2TimeStamp

The IBM DB2 UDB for iSeries supports three data types that deal with dates and times: DATE, TIME, and TIMESTAMP. The .NET Framework, however, has only a single built-in type for dates and times: DateTime. The DateTime data type has elements for both dates and times, including elements for the year, month, day, hour, minute, second, and millisecond. Because DateTime is the .NET Framework type most similar to DATE, TIME, and TIMESTAMP, these data types normally return their value as a DateTime, as shown in Table 4-5 on page 88. In this section we explain some of the issues you may see when using these data types.

When treating DATE, TIME, and TIMESTAMP fields as strings, the string value must be in one of the supported host ISO formats shown in Table 4-6.

Table 4-6 Supported DATE, TIME, and TIMESTAMP string formats

Data type	Supported string formats
Date	yyyy-mm-dd
Time	hh.mm.ss or hh:mm:ss
Timestamp	yyyy-mm-dd-hh.mm.ss.nnnnnn or yyyy-mm-dd hh:mm:ss.nnnnnn

When casting DATE, TIME, and TIMESTAMP fields to strings, cast them using a character that is the length of the resulting ISO string, as shown in Table 4-7.

Table 4-7 Casting DATE, TIME, and TIMESTAMP to strings

Data type	Sample cast statement for converting to string
Date	select cast(hiredate as char(10)) from sampledб.employee
Time	select cast(starting as char(8)) from sampledб.cl_sched
Timestamp	select cast(received as char(26)) from sampledб.in_tray

iDB2Date

iDB2Date corresponds to the iSeries DATE data type, which consists of a year, month, and day. Because the iSeries DATE does not include an hour, minute, second, or millisecond, a DateTime resulting from an iDB2Date always returns those elements initialized to zero.

iDB2Time

iDB2Time corresponds to the iSeries TIME data type, which consists of an hour, minute, and second. The hour is specified in 24-hour format, where hour 1 is 1:00 AM, hour 13 is 1:00 PM, and hour 0 represents 12:00 midnight. Because the iSeries TIME does not include a year, month, or day, a DateTime resulting from an iDB2Time always returns those elements initialized to a value of one. The DateTime's millisecond element is initialized to zero.

iDB2TimeStamp

iDB2TimeStamp corresponds to the iSeries TIMESTAMP data type, which consists of a year, month, day, hour, minute, second, and microsecond. The hour is specified in 24-hour format.

Because the iSeries TIMESTAMP can hold more precision (microseconds) than a DateTime (milliseconds), a DateTime resulting from an iDB2TimeStamp will have its millisecond element initialized to the iDB2TimeStamp's microseconds, truncated to three digits. Similarly, when initializing an iDB2TimeStamp using a DateTime, the microsecond element is initialized to the DateTime's millisecond element, multiplied by 1000.

Because the iDB2DataAdapter retrieves TIMESTAMP values as a DateTime, the three least significant digits of the TIMESTAMP value are lost. To avoid losing the significant digits, either cast the iDB2TimeStamp to a character string (see Example 4-77 on page 96), or use an iDB2DataReader instead of an iDB2DataAdapter, and read the timestamp values into an iDB2TimeStamp variable.

Example 4-76 shows how you can use a DataReader to read the timestamp value into an iDB2TimeStamp variable. For this example, we use the IN_TRAY table from the SAMPLEDB schema. We read the first field, RECEIVED, into an iDB2TimeStamp variable to make sure that we get the full microsecond precision. See 1.4, "DB2 UDB for iSeries sample schema" on page 8 for information about setting up the SAMPLEDB schema.

Example 4-76 Using an iDB2DataReader with GetiDB2TimeStamp to read a timestamp value

```
// Create and open a connection to the iSeries.
iDB2Connection cn = new iDB2Connection("DataSource=myiseries;");
cn.Open();

// Create a command which will select records
// from the IN_TRAY table.
iDB2Command cmd = new iDB2Command("select * from sampledby.in_tray", cn);

// Execute the query and get the results as a DataReader.
iDB2DataReader dr = cmd.ExecuteReader();

// Read the first field, RECEIVED, into an iDB2TimeStamp
// variable and also into a DateTime variable.
// Then print the results. You can see the iDB2TimeStamp
// returns greater precision than a DateTime returns.
while (dr.Read())
{
    iDB2TimeStamp mytimestamp = dr.GetiDB2TimeStamp(0);
    DateTime mydatetime = dr.GetDateTime(0);

    Console.WriteLine("iDB2TimeStamp: " + mytimestamp.ToNativeFormat() +
        ", DateTime: " + mydatetime.ToString());
}

// Close the DataReader since we no longer need it.
dr.Close();
```

```
// Dispose the command since we no longer need it.
cmd.Dispose();

// Close the connection.
cn.Close();
```

TIME and TIMESTAMP special values

With the IBM DB2 UDB for iSeries, you can insert TIME and TIMESTAMP values into a table using the special TIME value of 24:00:00 (or TIMESTAMP element containing a time of 24.00.00.000000). This special value is a non-standard way to indicate midnight. Note that even though the time values 24:00:00 and 00:00:00 are logically the same time, the two values do not compare the same. For more information, refer to the *DB2 Universal Database for iSeries SQL Reference*, which you can find in the iSeries Information Center at:

<http://www.iseries.ibm.com/infocenter>

Select Database → Reference → SQL Reference → Language elements → Assignments and comparisons → Datetime comparisons.

Because the .NET Framework DateTime type cannot handle an hour value of 24, using a TIME or TIMESTAMP that contains this value will cause your application to receive an *ArgumentOutOfRangeException*. To avoid this problem, you must cast the TIME or TIMESTAMP to a String value in your SELECT statement. Example 4-77 shows how to accomplish this. For this example, we use the IN_TRAY table from the SAMPLEDB. See 1.4, “DB2 UDB for iSeries sample schema” on page 8 for information about setting up the SAMPLEDB schema.

The CAST method also enables TIME and TIMESTAMP values to work with an iDB2DataAdapter. See Table 4-7 on page 94 for examples of how to cast DATE, TIME, and TIMESTAMP values.

Example 4-77 Using a CAST statement to cast a TIMESTAMP to a String

```
// Create and open a connection to the iSeries.
iDB2Connection cn = new iDB2Connection("DataSource=myiseries;");
cn.Open();

// Create a command which will select records
// from the IN_TRAY table.
iDB2Command cmd = cn.CreateCommand();
cmd.CommandText = "select cast(received as char(26)) from sampledb.in_tray";

// Execute the query and get the results as a DataReader.
iDB2DataReader dr = cmd.ExecuteReader();

while (dr.Read())
{
    // Display the data type of the RECEIVED field.
    // You can see the provider maps it to a String
    // because of the CAST statement.
    Console.WriteLine("Data type: " + dr.GetFieldType(0).ToString());

    // Display the field value.
    Console.WriteLine(", Value: " + dr.GetValue(0).ToString());
}

// Close the DataReader since we no longer need it.
dr.Close();
```

```
// Dispose the command since we no longer need it.
cmd.Dispose();

// Close the connection.
cn.Close();
```

Note: This example uses the IN_TRAY table, which includes a TIMESTAMP field. Although by default none of the data in the table contains a timestamp with a time value of 24:00:00, the example shows how you *could* retrieve that value if it exists in the table.

To write a TIME or TIMESTAMP value that contains the special 24:00:00 value, either write it using a string literal or initialize the parameter value using a string (Example 4-78). The parameter string value must be in a supported string format as shown in Table 4-6 on page 94.

Example 4-78 Updating a TIMESTAMP value using a parameterized UPDATE statement

```
// Create and open a connection to the iSeries.
iDB2Connection cn = new iDB2Connection("DataSource=myiseries;DefaultCollection=sampled;");
cn.Open();

// Create a command which will update the RECEIVED
// field for an entry in the IN_TRAY table.
iDB2Command cmd = cn.CreateCommand();
cmd.CommandText = "update in_tray set received=@newvalue where SOURCE='BADAMSON'";

// Derive the parameter information.
// It shows the @newvalue parameter is a DateTime.
cmd.DeriveParameters();
Console.WriteLine("Parameter data type: " + cmd.Parameters["@newvalue"].DbType);

// Set the parameter value to include a value that
// uses the special hour value of '24' for midnight.
// Because we set the value using a string, the data
// is not converted to a DateTime first.
cmd.Parameters["@newvalue"].Value = "2004-11-18-24.00.00.000000";
cmd.ExecuteNonQuery();

// Dispose the command since we no longer need it.
cmd.Dispose();

// Close the connection.
cn.Close();
```

iDB2Decimal and iDB2Numeric

The IBM DB2 UDB for iSeries supports two different decimal data types: *packed decimal*, which maps to iDB2Decimal, and *zoned decimal*, which maps to iDB2Numeric. For this section, we refer to them both as simply *decimal*. Decimal values on the iSeries can hold up to 63 decimal digits of precision. The decimal scale value can be any number between zero and the precision.

Note: The V5R3M0 release of DB2 UDB for iSeries increased support from 31 decimal digits of precision to 63 digits.

As shown in Table 4-5 on page 88, the IBM.Data.DB2.iSeries provider maps decimal data to the .NET Framework Decimal type. While this is normally the best choice, problems can occur because .NET's Decimal objects can hold only up to 28 or 29 digits of precision, less than the

maximum supported by the iSeries. If you have decimal data on the iSeries and read the value using a Decimal (which is the default), you could end up with a loss of precision (some digits following the decimal point are truncated) or an `OverflowException` (if the number of digits before the decimal point is greater than the 28 or 29 digits that Decimal supports).

To avoid these problems with decimal data, you have several options:

- ▶ Use a `DataReader` and call `GetString()` to retrieve the decimal value as a string.
- ▶ Use a `DataReader` and call `GetDB2Decimal` or `GetDB2Numeric`.
- ▶ Cast the decimal value using your `Select` statement to return the value as a string. This method works with both the `DataReader` and `DataAdapter`.

To illustrate these options, we create a table in our `SAMPLEDB` schema called `DEC` as shown in Example 4-79. See 1.4, “DB2 UDB for iSeries sample schema” on page 8 for a description of how to set up the `SAMPLEDB` schema.

Example 4-79 Creating the DEC table to hold large decimal data

```
// Create and open a connection to the iSeries.
iDB2Connection cn = new iDB2Connection("DataSource=myiseries;DefaultCollection=sampled;");
cn.Open();

// Create and run a command which will create the DEC table.
iDB2Command cmd = cn.CreateCommand();
cmd.CommandText = "create table dec (dec1 decimal(63, 62), num1 numeric(63, 3))";
cmd.ExecuteNonQuery();

// Insert some large decimal data into the table.
// We use a parameterized insert statement just for fun.
cmd.CommandText = "insert into dec values(@dec1, @num1)";
cmd.DeriveParameters();
cmd.Parameters["@dec1"].Value =
"1.123456789012345678901234567890123456789012345678901234567890123456789012";
cmd.Parameters["@num1"].Value =
"123456789012345678901234567890123456789012345678901234567890.123";
cmd.ExecuteNonQuery();

// Dispose the command since we no longer need it.
cmd.Dispose();

// Close the connection.
cn.Close();
```

Note: This example works with iSeries hosts V5R3M0 and later. Older hosts support only up to 31 digits of precision.

Now that we have a table containing large decimal values, we can retrieve the data. Example 4-80 shows how to read the decimal data using an `iDB2DataReader`. We show the `GetString()`, `GetDB2Decimal()`, and `GetDB2Numeric()` methods.

Example 4-80 Reading large decimal data using a DataReader

```
// Create and open a connection to the iSeries.
iDB2Connection cn = new iDB2Connection("DataSource=myiseries;DefaultCollection=sampled;");
cn.Open();

// Create a command which will select records from the DEC table.
iDB2Command cmd = new iDB2Command("select * from dec", cn);
```

```

// Execute the command and get a DataReader in return.
iDB2DataReader dr = cmd.ExecuteReader();

// Read the data using GetString, GetiDB2Decimal, and
// GetiDB2Numeric. Display the results as a string.
while (dr.Read())
{
    // Retrieve the data using several different methods
    String dec1String = dr.GetString(0);
    String num1String = dr.GetString(1);
    iDB2Decimal dec1Decimal = dr.GetiDB2Decimal(0);
    iDB2Numeric num1Numeric = dr.GetiDB2Numeric(1);

    // Display the data using several different methods
    Console.WriteLine("DEC1 values");
    Console.WriteLine("-----");
    Console.WriteLine("GetString: " + dec1String);
    Console.WriteLine("GetiDB2Decimal: " + dec1Decimal.ToString());

    Console.WriteLine("");
    Console.WriteLine("NUM1 values");
    Console.WriteLine("-----");
    Console.WriteLine("GetString: " + num1String);
    Console.WriteLine("GetiDB2Numeric: " + num1Numeric.ToString());
}

// Close the DataReader since we no longer need it.
dr.Close();

// Dispose the command since we no longer need it.
cmd.Dispose();

// Close the connection.
cn.Close();

```

Example 4-81 shows how to cast the decimal to a string using your Select statement. We cast the data using a char(64) because the decimal data consists of 63 digits plus a decimal separator. If your data could be negative, use an extra character to handle the sign character; for example, cast it as char(65). Casting the decimal/numeric like this is the only way to effectively handle large decimal/numeric data with an iDB2DataAdapter.

Example 4-81 Using a CAST statement to cast a decimal to a String

```

// Create and open a connection to the iSeries.
iDB2Connection cn = new iDB2Connection("DataSource=myiseries;DefaultCollection=sampled;");
cn.Open();

// Create a command which will select records from the DEC table.
iDB2Command cmd = cn.CreateCommand();
cmd.CommandText = "select cast(dec1 as char(64)), cast(num1 as char(64)) from dec";

// Execute the command and get a DataReader in return.
iDB2DataReader dr = cmd.ExecuteReader();

// Read the data using GetString.
while (dr.Read())
{
    // Retrieve the data as a string.
    String dec1String = dr.GetString(0);

```

```

String num1String = dr.GetString(1);

// Display the data
Console.WriteLine("DEC1 values");
Console.WriteLine("-----");
Console.WriteLine("GetString: " + dec1String);

Console.WriteLine("");
Console.WriteLine("NUM1 values");
Console.WriteLine("-----");
Console.WriteLine("GetString: " + num1String);
}

// Close the DataReader since we no longer need it.
dr.Close();

// Dispose the command since we no longer need it.
cmd.Dispose();

// Close the connection.
cn.Close();

```

Important: When casting the decimal/numeric to a character string as shown in the previous example, the decimal separator, whether a decimal point (.) or a comma (,), is *always* returned as a period (.), regardless of your thread's `CurrentCulture` setting. This is because the casting is done on the host server before the data is returned to the provider. The provider does not give you a way to set the decimal separator value used by the host server job servicing your SQL requests.

Writing decimal data can always be done in either of two ways:

- Using a literal value to insert the decimal data; for example:

```

insert into sampled.b.dec
values ('1.1234567890123456789012345678901234567890123456789012',
'123456789012345678901234567890123456789012345678901234567890.123')

```

- Assigning a decimal parameter using a string value, as shown in Example 4-79 on page 98

Decimal separator

Decimal and numeric data (`iDB2Decimal` and `iDB2Numeric`) normally contains a decimal separator character, usually either a period (.) or a comma (,) depending on your thread's `CurrentCulture` setting. The `IBM.Data.DB2.iSeries` provider handles the decimal separator automatically. It assumes that when you send decimal strings to the host using a parameterized insert or update statement, the decimal separator in the string is the decimal separator that corresponds to your thread's `CurrentCulture` setting. Similarly, when the provider returns a decimal string (for example, via the `DataReader`'s `GetString()` method, or the `iDB2Decimal` or `iDB2Numeric`'s `ToString()` method), the provider returns the correct decimal separator for your thread's `CurrentCulture` setting. The exception to this is when you cast the decimal data using your `Select` statement as shown in Example 4-81 on page 99, where the decimal separator is always returned as a period (.). This is because the casting is done on the host server before the data is returned to the provider, and the provider does not give you a way to set the decimal separator value used by the host server job servicing your SQL requests.

Example 4-82 is a simple example that shows how your decimal separator character is automatically handled by the provider. It does the following:

1. Sets the thread's `CurrentCulture` to `de-DE`, which uses a comma for the decimal separator character.
2. Creates a table in the `SAMPLEDB` schema called `DECIMALDE` containing a decimal and a numeric field.
3. Inserts data into the table using a parameterized insert statement. The inserted data is initialized using a string containing a comma for the decimal separator.
4. Reads the data from the table using a `DataReader`. The data comes back with the correct decimal separator.

Example 4-82 Using decimal data with a culture that uses a comma for the decimal separator

```
CultureInfo currCulturex = new CultureInfo("de-DE");
Thread.CurrentThread.CurrentCulture = currCulturex;

// Create and open a connection to the iSeries.
iDB2Connection cn = new iDB2Connection("DataSource=myiseries;DefaultCollection=sampled;");
cn.Open();

// Create a table called DECIMALDE.
iDB2Command cmd = cn.CreateCommand();
cmd.CommandText = "create table decimalde (dec1 decimal(5, 2), num1 numeric(4, 3))";
cmd.ExecuteNonQuery();

// Create a parameterized insert statement
cmd.CommandText = "insert into decimalde values(@dec1, @num1)";
cmd.DeriveParameters();

// Insert a row into the table.
// The string we use to initialize the decimal data
// uses a comma (',') for the decimal separator.
cmd.Parameters["@dec1"].Value = "123,45";
cmd.Parameters["@num1"].Value = "1,123";
cmd.ExecuteNonQuery();

// Now, read the data back using a DataReader.
cmd.CommandText = "select * from decimalde";
cmd.Parameters.Clear();
iDB2DataReader dr = cmd.ExecuteReader();

// Read the data using GetString, GetiDB2Decimal, and
// GetiDB2Numeric. Display the results as a string.
while (dr.Read())
{
    // Retrieve the data using several different methods
    String dec1String = dr.GetString(0);
    String num1String = dr.GetString(1);
    iDB2Decimal dec1Decimal = dr.GetiDB2Decimal(0);
    iDB2Numeric num1Numeric = dr.GetiDB2Numeric(1);

    // Display the data using several different methods
    Console.WriteLine("DEC1 values");
    Console.WriteLine("-----");
    Console.WriteLine("GetString: " + dec1String);
    Console.WriteLine("GetiDB2Decimal: " + dec1Decimal.ToString());

    Console.WriteLine("");
}
```

```

        Console.WriteLine("NUM1 values");
        Console.WriteLine("-----");
        Console.WriteLine("GetString: " + num1String);
        Console.WriteLine("GetIDB2Numeric: " + num1Numeric.ToString());
    }

    // Close the DataReader since we no longer need it.
    dr.Close();

    // Dispose the command since we no longer need it.
    cmd.Dispose();

    // Close the connection.
    cn.Close();

```

4.5.8 Handling exceptions

Most of the examples in this chapter optimistically assume that nothing will go wrong, so there is no exception handling in the examples. In a real-life .NET application, exception handling should be an integral part of your design. Exception handling is discussed at great length in the Microsoft .NET Framework documentation, available at:

<http://msdn.microsoft.com/library/>

Select **.NET Development** → **.NET Framework SDK** → **.NET Framework** → **Programming with the .NET Framework** → **Handling and Throwing Exceptions**.

The exceptions defined by the IBM.Data.DB2.iSeries provider are listed in 4.2.3, “Supported features” on page 36, and they are also described in the provider’s Technical Reference (4.4.1, “Displaying the technical reference” on page 41). The .NET provider’s exceptions inherit from `SystemException`.

When using any .NET code that could throw an exception, it is wise to surround the code with a try/catch block. The IBM.Data.DB2.iSeries provider may throw an exception at any time, for conditions such as an invalid `ConnectionString` value, a SQL error resulting from a non-existent table, of a communication error received because your host server job ended abnormally. When your application receives an exception, if you do not have a catch block to handle that exception, a message is displayed by the Microsoft .NET runtime. For exceptions that inherit from `SystemException` (as all of the provider’s exceptions inherit from `SystemException`), the message you see simply says `System error`. This is not very helpful. If you instead catch the exception, you can examine the exception object and get more details about the cause of the error. At the very least, you should examine the exception’s `Message` property, because it often helps pinpoint the cause of an error.

The most common provider-specific exceptions you are likely to receive are the `iDB2SQLException` and the `iDB2CommErrorException`. First, we show an example of coding to handle an `iDB2SQLException`, then we discuss `iDB2CommErrorException` in greater detail.

Handling SQL errors (`iDB2SQLException`)

A SQL error can happen when you prepare, execute, or derive parameter information for a command. When an application receives a SQL error, the following properties of the `iDB2SQLException` object can help you gather more information about the error:

- ▶ **Message** property: The `Message` property contains a description of the SQL error.
- ▶ **MessageDetails** property: The `MessageDetails` property contains a detailed description of the error, typically including the cause and recovery.

- ▶ **MessageCode** property: The **MessageCode** property contains a number that corresponds to the SQL error you receive.
- ▶ **SqlState** property: The **SqlState** property contains the SQLSTATE returned from the host. **SqlState** is valid only for SQL errors.

Example 4-83 shows an example of a SQL0204 error. We force this error by executing a Select statement for a table that does not exist.

*Example 4-83 Handling an **iDB2SQLException***

```
// Create and open a connection to the iSeries
iDB2Connection cn = new
iDB2Connection("DataSource=myiseries;DefaultCollection=sampladb;");
cn.Open();

// Create a command that selects from a non-existent table.
// We expect this to fail with a SQL0204 error.
iDB2Command cmd = new iDB2Command("select * from notthere", cn);
try
{
    iDB2DataReader dr = cmd.ExecuteReader();
}
catch (iDB2SQLException e)
{
    Console.WriteLine("Message: " + e.Message);
    Console.WriteLine("MessageCode: " + e.MessageCode.ToString());
    Console.WriteLine("SqlState: " + e.SqlState);
    Console.WriteLine("");
    Console.WriteLine("MessageDetails: " + e.MessageDetails);
}
```

When you run Example 4-83, the **Message** property tells the error, and the **MessageCode** property contains a number that corresponds to the message. In this example, **MessageCode** is -204, which corresponds to the SQL error SQL0204. For more information about any SQL error, you can display the message description on your iSeries with this command:

```
DSPMSGD RANGE(SQL0204) MSGF(QSQLMSG)
```

Simply replace 0204 (without the negative sign) with the four-character **MessageCode** number.

Handling communication errors (**iDB2CommErrorException**)

When you execute commands using **IBM.Data.DB2.iSeries**, the provider uses a communication link to transfer the commands and data back and forth to the iSeries server job that runs requests on behalf of your application. At times, this communication link may become unusable for any of several reasons, including:

- ▶ The iSeries server is IPLed (for example, to perform nightly maintenance).
- ▶ The iSeries server job processing your requests (QZDASOINIT) is ended.
- ▶ The communication link experiences some other failure.

Whatever the cause, your application should be prepared to handle communication errors whenever it executes commands. Example 4-84 on page 104 shows how to catch the **iDB2CommErrorException**. In our example, we force a communication error by asking to terminate the host server job using the **ENDJOB** command. We then simply display the error information when the communication exception is caught. Before running this console application, add a reference to **System.Windows.Forms** (**Project** → **Add Reference** → **.NET tab** → **System.Windows.Forms.dll**).

Example 4-84 Handling an iDB2CommErrorException

```
// Create and open a connection to the iSeries
iDB2Connection cn = new
iDB2Connection("DataSource=myiSeries;DefaultCollection=sampledb;");
cn.Open();

// Now that we have an open connection, kill the host server job.
String msg = "Please run this command to end the host server job:\r\n" +
    "endjob job(" + cn.JobName.Trim() + ") option(*immed)";
System.Windows.Forms.MessageBox.Show(msg, "Before you press OK...");

// Create a command that selects from a table.
// Because we killed the host server job, this command
// will fail with iDB2CommErrorException.
iDB2Command cmd = new iDB2Command("select * from employee", cn);
try
{
    iDB2DataReader dr = cmd.ExecuteReader();
}
catch (iDB2CommErrorException e)
{
    Console.WriteLine("Message: " + e.Message);
    Console.WriteLine("MessageCode: " + e.MessageCode.ToString());
    Console.WriteLine("MessageDetails: " + e.MessageDetails);
}
```

For the `iDB2CommErrorException`, the `Message` property says that a communication failure occurred, and the `MessageDetails` property tells what type of communication failure it was. In some cases, the `MessageDetails` contains a message number, such as CWBCO1047. Read more about this and other messages and their possible causes in the *iSeries Access for Windows User's Guide*. To display the User's Guide from the Windows desktop, select **Start** → **Programs** → **IBM iSeries Access for Windows** → **User's Guide**. From the User's Guide, click the **Contents** tab and expand **Messages** → **iSeries Access for Windows messages**. Then scroll through the list of messages until you find the one you are looking for.

The `MessageCode` property contains a return code value that may be useful if a persistent problem is reported to IBM Service.

Recovering from a communication error

If your application routinely keeps connections active during periods when your host server connections may be lost, you should add extra processing to recover from the communication failures that result. Often, all you have to do is close and then reopen your connection. Example 4-85 shows how to modify the previous example to close and then reopen the connection when the communication failure occurs.

Example 4-85 Recovering from an iDB2CommErrorException by closing and reopening the connection

```
// Create and open a connection to the iSeries
iDB2Connection cn = new
iDB2Connection("DataSource=myiseries;DefaultCollection=sampledb;");
cn.Open();

// Now that we have an open connection, kill the host server job.
String msg = "Please run this command to end the host server job:\r\n" +
    "endjob job(" + cn.JobName.Trim() + ") option(*immed)";
System.Windows.Forms.MessageBoxButtons.OK);
System.Windows.Forms.MessageBox.Show(msg, "Before you press OK...");
```

```

// Create a command that selects from a table.
// Because we killed the host server job, this command
// will fail with iDB2CommErrorException.
iDB2Command cmd = new iDB2Command("select * from employee", cn);
try
{
    iDB2DataReader dr = cmd.ExecuteReader();
}
catch (iDB2CommErrorException e)
{
    Console.WriteLine("Message: " + e.Message);
    Console.WriteLine("MessageCode: " + e.MessageCode.ToString());
    Console.WriteLine("MessageDetails: " + e.MessageDetails);

    // Now, recover from the failure by closing and then
    // reopening the connection.
    cn.Close();
    cn.Open();
}

// If we get to here, we should have an active connection again.
// Try the command again.
iDB2DataReader dr2 = cmd.ExecuteReader();

// Close the DataReader since we're done with it.
dr2.Close();

// Dispose the command since we're done with it.
cmd.Dispose();

// Close the connection once more.
cn.Close();

```

Special considerations when using connection pooling

By default, the IBM.Data.DB2.iSeries provider enables connection pooling, in which a pool of connections to iSeries host server jobs are maintained by the provider. When you open a pooled connection (`iDB2Connection.Open` method), one of these pooled iSeries server connections is removed from the pool and associated with your `iDB2Connection` object. When you close the pooled connection (`iDB2Connection.Close` method), the iSeries server connection is released back into the pool to be reused.

Because a pooled connection keeps an active connection to the iSeries server, it is possible that the communication link could be lost while the pooled connection is sitting idle in the pool. For example, this scenario could occur if you keep your application running overnight with pooled connections still active, and your iSeries server has maintenance performed on it, causing all of the server jobs to end. The next day when you open a connection through your application, a pooled connection is handed to you that does not work because the server job has ended. This error is not detected until you try to use the connection.

Example 4-86 on page 106 shows this scenario. As in the previous example, we force an end to the host server job. This time, the job we end is a pooled connection. The communication error is detected only when we try to run a command against that connection. Just as in our previous example, we recover from the error by closing and then reopening the connection.

Example 4-86 Recovering from an *IDB2CommErrorException* on a pooled connection

```
// Create and open a connection to the iSeries.
iDB2Connection cn = new
iDB2Connection("DataSource=myiseries;DefaultCollection=sampledb;");
cn.Open();

// Create a message containing the server job information.
// We'll display this message after we close the connection.
String msg = "Please run this command to end the host server job:\r\n" +
    "endjob job(" + cn.JobName.Trim() + ") option(*immed)";

// Close the connection so it is released back into
// the connection pool.
cn.Close();

// Now that the connection is in the pool, kill the host server job.
System.Windows.Forms.MessageBox.Show(msg, "Before you press OK...");

// Now, open the connection again. This will cause the
// now dead connection to be handed back. Note the
// Open() does not return an exception.
cn.Open();

// Create a command that selects from a table.
// Because we killed the host server job, this command
// will fail with iDB2CommErrorException.
iDB2Command cmd = new iDB2Command("select * from employee", cn);
try
{
    iDB2DataReader dr = cmd.ExecuteReader();
}
catch (iDB2CommErrorException e)
{
    Console.WriteLine("Message: " + e.Message);
    Console.WriteLine("MessageCode: " + e.MessageCode.ToString());
    Console.WriteLine("MessageDetails: " + e.MessageDetails);

    // Now, recover from the failure by closing and then
    // reopening the connection.
    cn.Close();
    cn.Open();
}

// If we get to here, we should have an active connection again.
// Try the command again.
iDB2DataReader dr2 = cmd.ExecuteReader();

// Close the DataReader since we're done with it.
dr2.Close();

// Dispose the command since we're done with it.
cmd.Dispose();

// Close the connection once more.
cn.Close();
```

Persistent communication errors

Example 4-85 on page 104 and Example 4-86 showed how to recover from an *IDB2CommErrorException* by closing then reopening the connection. Although this method

works in most cases, you may still receive an `iDB2CommErrorException` when you reopen the connection. This could happen for any of several reasons, including:

- ▶ The pooled connections no longer have host server jobs associated with them if the iSeries server was IPLed or the server jobs' subsystem was ended and then restarted.
- ▶ The iSeries server is not active, or the server jobs' subsystem is not active.
- ▶ Numerous other causes of communication errors.

You can avoid repeated communication errors for the first condition by using the `CheckConnectionOnOpen` property in your `ConnectionString`. See "CheckConnectionOnOpen" on page 55 for more information about this property.

Note: V5R3M0 added the `CheckConnectionOnOpen` property in service pack SI17742.

Sometimes communication errors continue due to an unresolved problem. When this happens, your application should gracefully continue or exit.

4.6 Common tasks

In this section, we discuss some common scenarios for using the `IBM.Data.DB2.iSeries` provider and provide coding samples.

4.6.1 A `DataReader` example

Many of the examples shown previously in this chapter use an `iDB2DataReader` to read values from a table. In this section, we illustrate the `iDB2DataReader` again, using the `EMPLOYEE` table from the `SAMPLEDB` schema. See 1.4, "DB2 UDB for iSeries sample schema" on page 8 to set up the `SAMPLEDB` schema.

We show two `DataReader` examples. Example 4-87 selects records from the `EMPLOYEE` table, reads each value, and prints the value to the console window. We show two ways to retrieve a `DATE` field: `Hiredate` is retrieved using `GetiDB2Date()` and `Birthdate` is retrieved using `GetDateTime()`. The second example, Example 4-88 on page 109, selects the same records from the `EMPLOYEE` table. This time, the data is added to a `DataTable`, and the `DataTable` is displayed in a `DataGrid` using a Windows application.

In 4.6.2, "A simple `DataAdapter` with `CommandBuilder` example" on page 110, we show how to use a `DataAdapter` to accomplish the same task.

A Console `DataReader` example

For Example 4-87, we use a Console application to read from the `EMPLOYEE` table.

Example 4-87 Using a `DataReader` to read records from the `EMPLOYEE` table and print the values

```
// Create and open a connection to the iSeries.
iDB2Connection cn = new iDB2Connection("DataSource=myiseries;DefaultCollection=sampledb;");
cn.Open();

// Create a command that selects records from the EMPLOYEE table.
iDB2Command cmd = new iDB2Command("select * from employee", cn);

// Execute the command, and get a DataReader in return.
iDB2DataReader dr = cmd.ExecuteReader();
```

```
// Read each row from the table and display the information.
int i = 1;

while (dr.Read())
{
    Console.WriteLine("Row " + i.ToString() + ":");
    Console.WriteLine("-----");
    Console.WriteLine("Employee {0}: {1} {2} {3}",
        dr.GetString(0), dr.GetString(1).TrimEnd(),
        dr.GetChar(2), dr.GetString(3).TrimEnd());
    Console.WriteLine("Workdept: {0}, Phone#: {1}, Hiredate: {2}",
        dr.GetString(4), dr.GetString(5), dr.GetDateTime(6).ToString());
    Console.WriteLine("Job: {0}, Education level: {1}, Sex: {2}, Birthdate: {3}",
        dr.GetString(7).TrimEnd(), dr.GetInt16(8), dr.GetChar(9),
        dr.GetDateTime(10).ToString());
    Console.WriteLine("Salary: {0}, Bonus: {1}, Commission: {2}",
        dr.GetDecimal(11), dr.GetDecimal(12), dr.GetDecimal(13));

    Console.WriteLine("");
    i++;
}

// Close the DataReader since we no longer need it.
dr.Close();

// Dispose the command since we no longer need it.
cmd.Dispose();

// Close the connection.
cn.Close();
```

A Windows DataReader example

For our next example, we use a Windows application to show how to use a DataReader and display the results in a DataGrid. For this example, create a Windows application instead of a Console application, as shown in Figure 4-16.

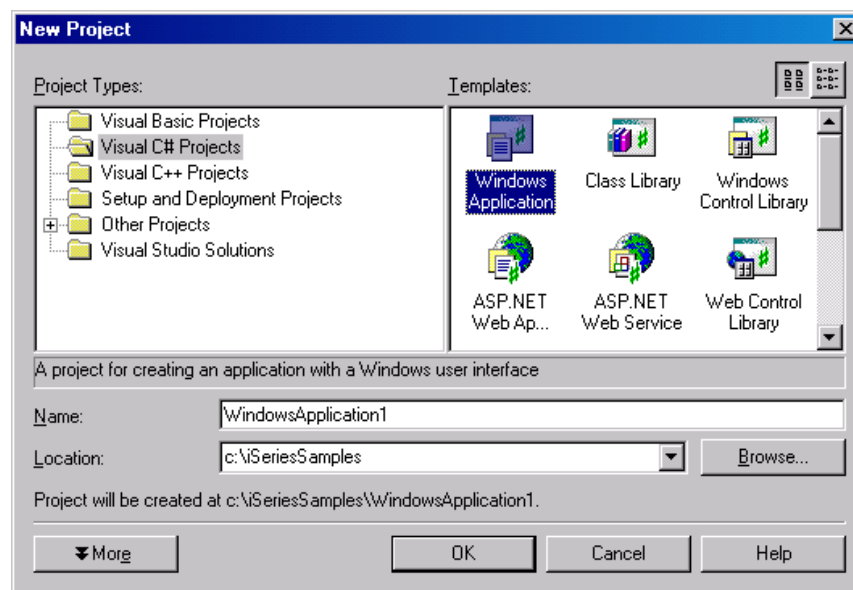


Figure 4-16 Create a C# Windows application

Be sure to add an assembly reference to the provider (see 4.4.3, “Adding an assembly reference to the provider” on page 43), and add a namespace directive to your C# source file (see 4.4.4, “Adding a namespace directive” on page 44). Next, double-click in any blank space on your Windows form. This causes Visual Studio .NET to create a method called `Form1_Load`. Enter the code shown in Example 4-88 into the `Form1_Load` method. For this example, for simplicity we show only the first six columns of the `EMPLOYEE` table.

Example 4-88 Using a `DataReader` to read records from the `EMPLOYEE` table into a `DataGrid`

```
// Create a DataGrid and add it to our form.
DataGrid datagrid = new DataGrid();
datagrid.Location = new Point(0, 0);
datagrid.Size = new Size(900, 900);

// Add the DataGrid to the form
Controls.AddRange(new Control[] {datagrid});

// Create a DataSet to hold our data.
DataSet ds = new DataSet();

// Create a DataTable and add a column
// for each column we select from our iSeries EMPLOYEE table.
// To make this example simple, we only display the first six fields:
// EMPNO, FIRSTNME, MIDINIT, LASTNAME, WORKDEPT, and PHONENO.
DataTable dt = new DataTable("employee");
dt.Columns.Add("empno");
dt.Columns.Add("firstnme");
dt.Columns.Add("midinit");
dt.Columns.Add("lastname");
dt.Columns.Add("workdept");
dt.Columns.Add("phoneno");

// Add the DataTable to the DataSet.
ds.Tables.Add(dt);

// Open a connection to the iSeries
iDB2Connection cn = new iDB2Connection("DataSource=myiseries;DefaultCollection=sampladb;");
cn.Open();

// Create a command that selects records from the EMPLOYEE table
iDB2Command cmd = new iDB2Command("select * from employee", cn);

// Execute the command, and get a DataReader in return.
iDB2DataReader dr = cmd.ExecuteReader();

// Read each row from the table, and put the results
// into the DataSet we created earlier.
while (dr.Read())
{
    // Create a new row to hold our data
    DataRow datarow = ds.Tables["employee"].NewRow();

    // Set the data into the row.
    // To make our example simple, we only show the
    // first six fields from the EMPLOYEE table.
    datarow["empno"] = dr.GetString(0);
    datarow["firstnme"] = dr.GetString(1).Trim();
    datarow["midinit"] = dr.GetChar(2);
    datarow["lastname"] = dr.GetString(3).Trim();
    datarow["workdept"] = dr.GetString(4);
}
```

```

        datarow["phoneno"] = dr.GetString(5);

        // Add the row to our DataTable
        ds.Tables["employee"].Rows.Add(datarow);
    }

    // Close the DataReader since we no longer need it.
    dr.Close();

    // Dispose the command since we no longer need it.
    cmd.Dispose();

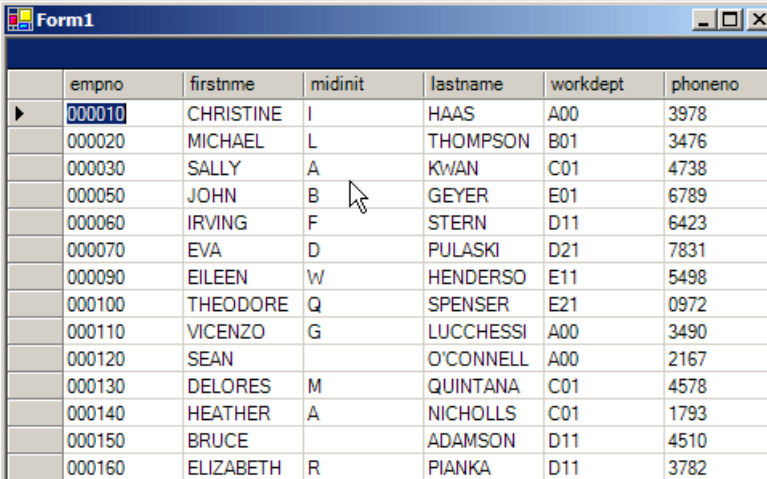
    // Close the connection.
    cn.Close();

    // Now, tell the DataGrid where the source data comes from.
    // The data comes from the DataSet we filled in earlier.
    datagrid.DataSource = ds;
    datagrid.DataMember = ds.Tables[0].TableName;

    // Finally, show the DataGrid.
    datagrid.Show();

```

Save your application (**File** → **Save All**) and run it. A form appears, showing the data grid filled in with the information from the EMPLOYEE table (Figure 4-17).



empno	firstme	midinit	lastname	workdept	phoneno
000010	CHRISTINE	I	HAAS	A00	3978
000020	MICHAEL	L	THOMPSON	B01	3476
000030	SALLY	A	KWAN	C01	4738
000050	JOHN	B	GEYER	E01	6789
000060	IRVING	F	STERN	D11	6423
000070	EVA	D	PULASKI	D21	7831
000090	EILEEN	W	HENDERSO	E11	5498
000100	THEODORE	Q	SPENSER	E21	0972
000110	VICENZO	G	LUCCHESSI	A00	3490
000120	SEAN		O'CONNELL	A00	2167
000130	DELORES	M	QUINTANA	C01	4578
000140	HEATHER	A	NICHOLLS	C01	1793
000150	BRUCE		ADAMSON	D11	4510
000160	ELIZABETH	R	PIANKA	D11	3782

Figure 4-17 The DataGrid shows the data we read from the EMPLOYEE table

4.6.2 A simple DataAdapter with CommandBuilder example

Although the DataReader is often the best way to read your data, sometimes a DataAdapter is the right choice. In this section, we show how to fill a DataGrid on a window using a few simple steps. The example shown in this section is also used as a starting point for the example shown in 4.7.3, “Updating DataSets” on page 136.

To read data from the EMPLOYEE table using a DataAdapter and CommandBuilder:

1. Create a new Windows application (see Figure 4-16 on page 108 for an example of how to do this). Add an assembly reference to the IBM.Data.DB2.iSeries provider (see 4.4.3, “Adding an assembly reference to the provider” on page 43), and add a namespace directive to your project (see 4.4.4, “Adding a namespace directive” on page 44).

2. Display the Toolbox component (**View** → **Toolbox**). Click **Data** to display the data access items. You should see a window that looks similar to Figure 4-18.

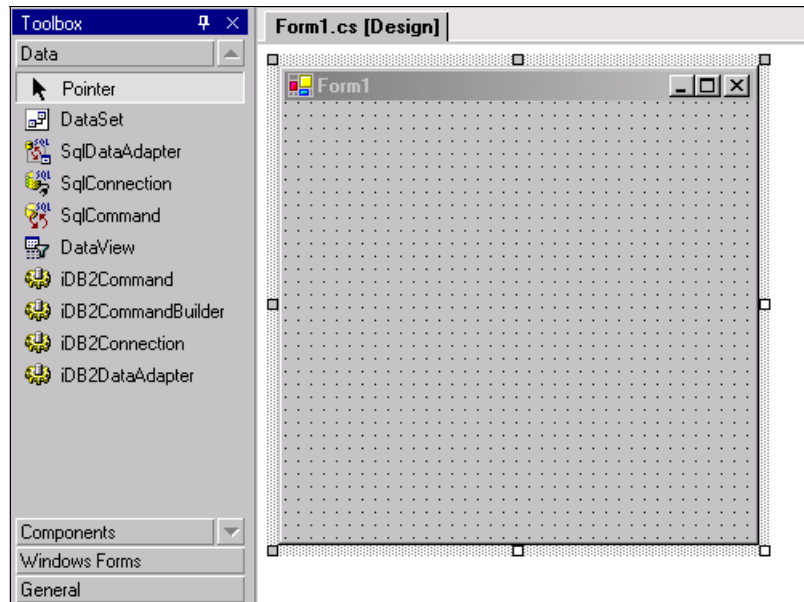


Figure 4-18 Display the Data Toolbox components

Note: If the data access component does not show iDB2Command, iDB2CommandBuilder, iDB2Connection, and iDB2DataAdapter, add them to the Toolbox manually: Select **Tools** → **Customize Toolbox** (or **Tools** → **Add/Remove Toolbox Items** in Visual Studio .NET 2003). Click the **.NET Framework Components** tab and scroll to the iDB2* components. Check the box next to iDB2Command, iDB2CommandBuilder, iDB2Connection, and iDB2DataAdapter, then click **OK**.

3. Drag data access components to your window. From the Toolbox, drag **iDB2Connection**, **iDB2Command**, **iDB2CommandBuilder**, and **iDB2DataAdapter** to your window. The items will appear below the window (Figure 4-19 on page 112).

Note: This example includes a CommandBuilder, even though it is not used here, because it will be used in 4.7.3, “Updating DataSets” on page 136 when we discuss updatable DataSets.

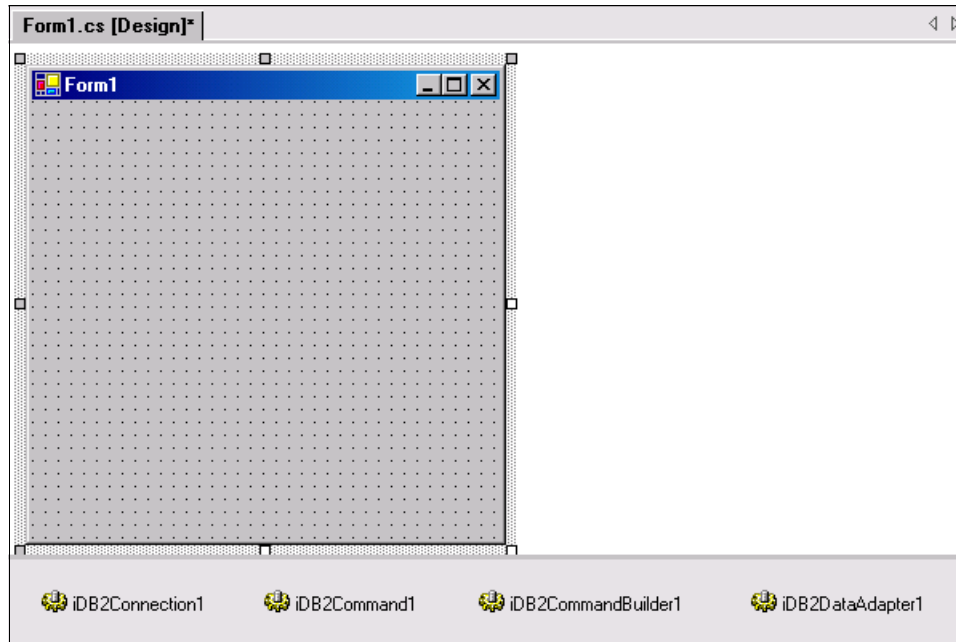


Figure 4-19 Drag the iDB2* components to your window

4. Drag a DataSet to your window. On the Add Dataset window, select **Untyped dataset** and click **OK** (Figure 4-20).

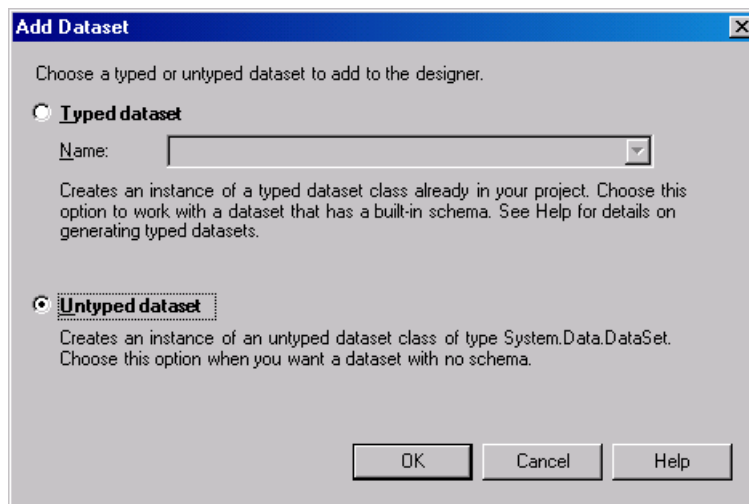


Figure 4-20 Drag an Untyped DataSet to your window

5. Add a DataGrid to your window: Select the **Windows Forms** Toolbox item. Drag a DataGrid to your window, and size the DataGrid to fill up most of the window. You can size the window larger if you wish. Your window should look similar to Figure 4-21.

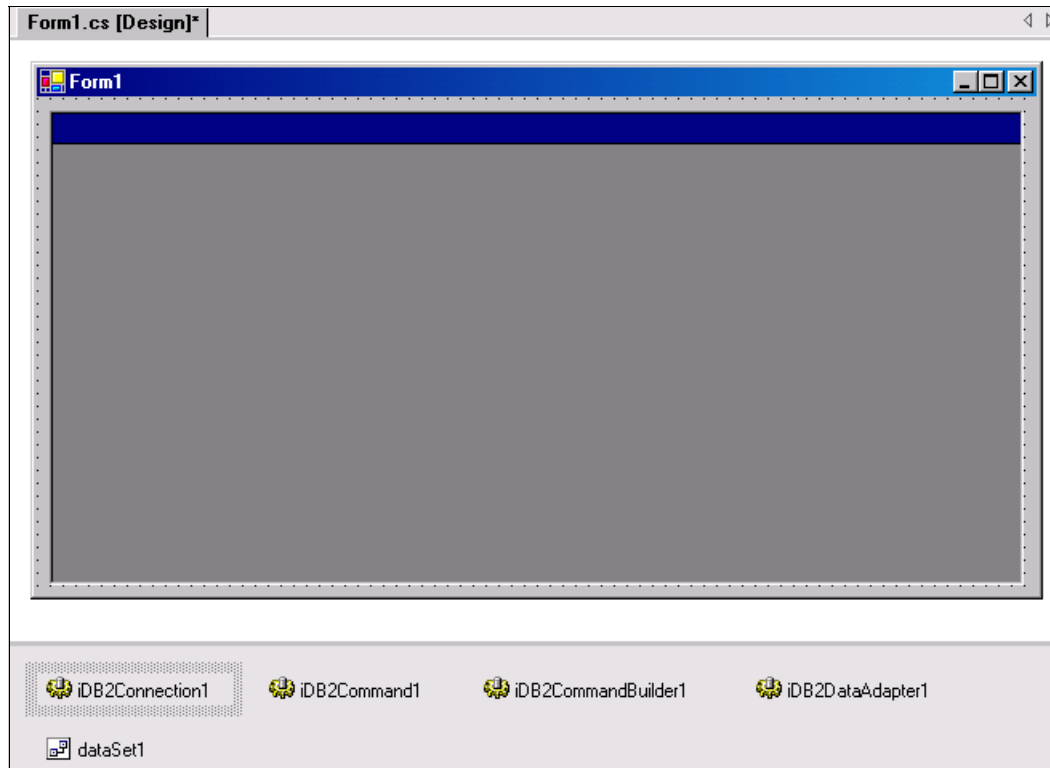


Figure 4-21 Add a DataGrid to your window

6. Set the object properties: Right-click **iDB2Connection1** and select **Properties**. In the Properties panel, select the box next to **ConnectionString** and type in the following ConnectionString:

DataSource=myiseries; DefaultCollection=sampled;

Press Enter. The Properties panel is updated to reflect your DataSource and DefaultCollection as shown in Figure 4-22.

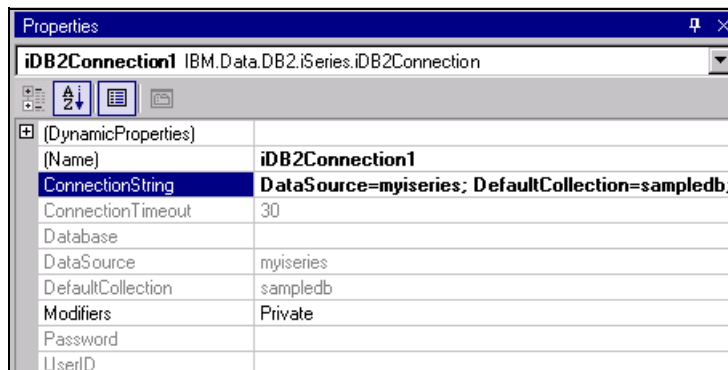


Figure 4-22 Set the iDB2Connection1 object properties

7. Set the iDB2Command1 properties: Click **iDB2Command1**. In the Properties panel, select the box next to **CommandText** and type in:

select * from employee

8. Select the box next to Connection. In the pull-down menu, select **iDB2Connection1**. The Properties panel should look similar to Figure 4-23.

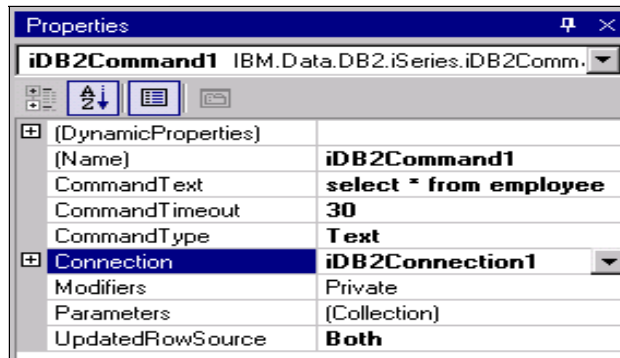


Figure 4-23 Set the iDB2Command1 object properties

9. Update the DataAdapter properties: Click **iDB2DataAdapter1**. In the Properties panel, select the box next to SelectCommand. In the pull-down menu, select **iDB2Command1** as shown in Figure 4-24.

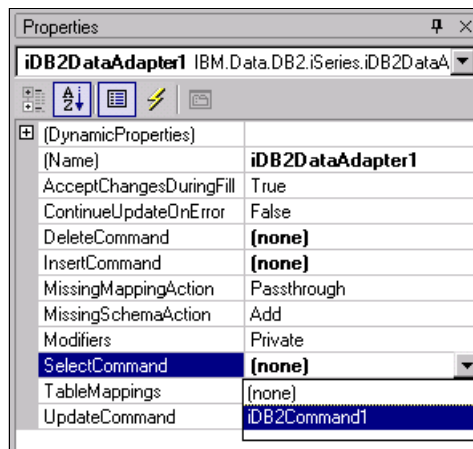


Figure 4-24 Set the iDB2DataAdapter1 object properties

10. Update the CommandBuilder properties: Click **iDB2CommandBuilder1**. In the Properties panel, select the box next to DataAdapter. In the pull-down menu, select **iDB2DataAdapter1** as shown in Figure 4-25.

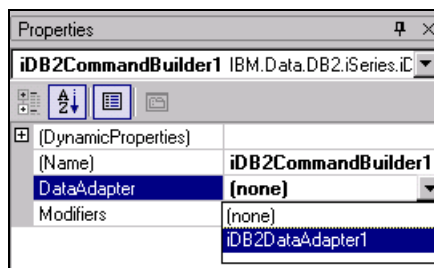


Figure 4-25 Set the iDB2CommandBuilder1 object properties

11. Update the DataGrid properties: Click the DataGrid object. In the Properties panel, select the box next to DataSource. In the pull-down menu, select **dataSet1** as shown in Figure 4-26 on page 115.

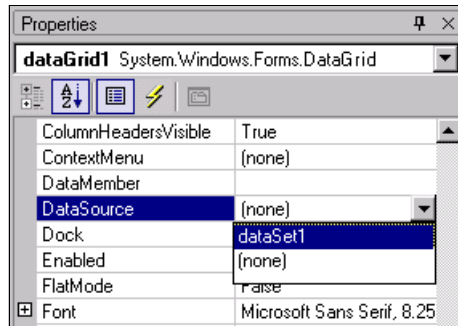


Figure 4-26 Set the dataGrid1 object properties

12. Write code to tie it all together. So far in this example, you have not written any code. Now with only a few lines of code, we fill the DataSet using the DataAdapter, then display the DataGrid containing our data. On your Form1 design window, double-click somewhere in the window (not on the DataGrid). Visual Studio .NET creates a method called Form1_Load and places the cursor at the top of this method. Add the code in Example 4-89 to the Form1_Load routine as shown in Figure 4-89.

Example 4-89 Fill the DataSet using our DataAdapter

```
iDB2DataAdapter1.Fill(dataSet1);
dataGrid1.DataMember = dataSet1.Tables[0].TableName;
```

When you call the DataAdapter's Fill() method, the DataAdapter opens the connection, fills the DataSet, and closes the connection.

13. Run the application.

Save the application (**File** → **Save All**), then run it. A window appears with the DataGrid filled with data read from the EMPLOYEE table, as shown in Figure 4-27.

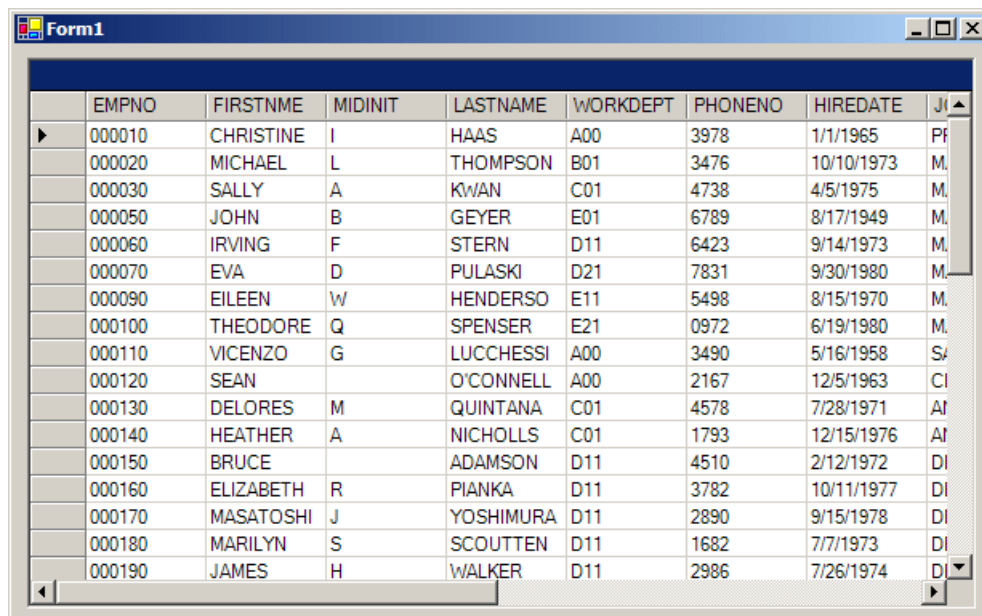


Figure 4-27 The DataGrid shows the data we read from the EMPLOYEE table

This section gives a quick, easy way to display data in a DataGrid. For a more advanced example that allows updates, see 4.7.3, "Updating DataSets" on page 136.

4.6.3 Using transactions

Transactions enable you to group statements together and execute all or none of them. If all statements in the transaction are successful, they can be committed to the database (made permanent). If one or more statements in the transaction fail, they can be rolled back to pre-transaction state. A typical transaction scenario uses a debit/credit example. When a customer transfers funds from one bank account to another, the entire transfer is contained in a single transaction. However, if money is withdrawn from the first account but an error occurs and the money cannot be deposited into the second account, the transaction is rolled back and the entire transfer is considered a failure. The entire transaction will be committed only if the withdrawal from the first account *and* the deposit into the second account are successful. Transaction processing is sometimes referred to as *commitment control* because transactions enable you to control when or if changes are committed to the database.

To use transactions, the table (or tables) that are used within a transaction must be *journalled*. Journaling is the process that enables transactions to be committed or rolled back. Read more about journaling at the IBM Information Center at:

<http://www.iseries.ibm.com/infocenter>

Select **Database** → **Programming** → **SQL Programming** → **Data Protection** → **Data integrity** → **Journaling**.

For more information about journaling performance, refer to the redbook *Striving for Optimal Journal Performance on DB2 Universal Database for iSeries*, SG246286.

Local transactions

The IBM.Data.DB2.iSeries provider supports local transactions. Local transactions are performed entirely as a single unit of work on a single connection. If your application does not require a transaction to span across multiple iSeries connections or across multiple databases, use local transactions because they provide the best performance.

To begin a local transaction, call the iDB2Connection object's BeginTransaction method. To permanently commit changes made during the transaction to the database, use the transaction's Commit() method. To cancel changes made during the transaction, use the transaction's Rollback() method. Example 4-90 shows a sample application that uses a local transaction to increase an employee's salary and reduce the employee's bonus by the same amount. It calls BeginTransaction to start the transaction, and receives an iDB2Transaction object in return. Commands are executed within the transaction, surrounded by a try/catch block. If the commands execute without error, the changes are committed to the database. If an exception causes a command to fail, the changes are rolled back and an error message is displayed. This example uses the EMPLOYEE table in the SAMPLEDB schema. See 1.4, "DB2 UDB for iSeries sample schema" on page 8 about setting up the sample schema.

Example 4-90 Using a transaction to change an employee's salary and bonus

```
// Create and open a connection to the iSeries.
iDB2Connection cn = new iDB2Connection("DataSource=myiseries;DefaultCollection=sampledb;");
cn.Open();

// Begin a transaction on this connection.
iDB2Transaction trans = cn.BeginTransaction(IsolationLevel.RepeatableRead);

// Create a couple of commands which will run under this transaction.
iDB2Command increaseSalary = cn.CreateCommand();
iDB2Command decreaseBonus = cn.CreateCommand();

// Set the command text for our commands
```

```

increaseSalary.CommandText = "update employee set salary = salary + 500 where job='PRES'";
decreaseBonus.CommandText = "update employee set bonus = bonus - 500 where job='PRES'";

// Execute the commands within the transaction.
// We surround the commands with an exception handler,
// to ensure we can rollback the transaction if
// something goes wrong.
try
{
    increaseSalary.ExecuteNonQuery();
    decreaseBonus.ExecuteNonQuery();

    // If we get to here without an exception, we know both
    // commands were successful. Commit the transaction.
    trans.Commit();
    Console.WriteLine("The transaction was performed successfully.");
}
catch
{
    // If we get to here, something bad happened.
    // Rollback the transaction.
    trans.Rollback();
    Console.WriteLine("An error occurred. The transaction was rolled back.");
}

// Dispose the transaction since it is no longer useful.
trans.Dispose();

// Dispose the commands since we no longer need them.
increaseSalary.Dispose();
decreaseBonus.Dispose();

// Close the connection.
cn.Close();

```

After a connection's `BeginTransaction` method is called to start a transaction, the connection is in transaction mode. All commands that execute under that connection must then run under the transaction that was started by the connection. To accomplish this, the `iDB2Connection`'s `Transaction` property must match the `iDB2Transaction` returned from `BeginTransaction`. There are several ways to set the command's `Transaction` property:

1. Automatically, by using the `iDB2Connection`'s `CreateCommand()` method to create the command object. If the connection has already started a transaction, any commands created using `CreateCommand()` automatically run under the same transaction. This method is shown in Example 4-91.

Example 4-91 Setting the Transaction property automatically

```

// Create and open a connection to the iSeries.
iDB2Connection cn = new iDB2Connection("DataSource=myiseries;DefaultCollection=sampldb;");
cn.Open();

// Begin a transaction on this connection.
iDB2Transaction trans = cn.BeginTransaction(IsolationLevel.RepeatableRead);

// Create a command using CreateCommand().
// The new command automatically runs under the
// 'trans' transaction we started above.
iDB2Command cmd = cn.CreateCommand();

```

2. Manually assign the command's Transaction property to the iDB2Transaction returned from BeginTransaction. (Example 4-92).

Example 4-92 Setting the Transaction property manually

```
// Create and open a connection to the iSeries.  
iDB2Connection cn = new iDB2Connection("DataSource=myiseries;DefaultCollection=samledb;");  
cn.Open();  
  
// Begin a transaction on this connection.  
iDB2Transaction trans = cn.BeginTransaction(IsolationLevel.RepeatableRead);  
  
// Create a command using the new operator.  
iDB2Command cmd = new iDB2Command();  
  
// Manually set the command's Connection and  
// Transaction properties.  
cmd.Connection = cn;  
cmd.Transaction = trans;
```

3. Pass the iDB2Transaction returned from BeginTransaction to the iDB2Command's constructor when you create the command object (Example 4-93).

Example 4-93 Setting the Transaction property when the command is constructed

```
// Create and open a connection to the iSeries.  
iDB2Connection cn = new iDB2Connection("DataSource=myiseries;DefaultCollection=samledb;");  
cn.Open();  
  
// Begin a transaction on this connection.  
iDB2Transaction trans = cn.BeginTransaction(IsolationLevel.RepeatableRead);  
  
// Create a command using the new operator.  
// Pass the command text, the connection, and  
// the transaction to the constructor.  
iDB2Command cmd = new iDB2Command("select * from employee", cn, trans);
```

After a transaction is committed or rolled back, the connection reverts to autocommit mode, where all updates to the database are made permanent without the ability to commit or roll back. See "Autocommit" on page 119.

Distributed transactions and automatic transaction enlistment

Distributed transactions enable transactions to span multiple iSeries connections or multiple heterogeneous databases. Distributed transactions are managed by a *distributed transaction coordinator*, typically a third-party software package. They require each database provider to support a two-phase commit model. As of this writing, Microsoft has not provided a distributed transaction coordinator that runs as a managed .NET application. The most commonly used transaction coordinator is the COM-based Distributed Transaction Coordinator, also called DTC, from Microsoft.

The IBM.Data.DB2.iSeries provider does not currently support distributed transactions. If your application needs the ability for a transaction to span multiple iSeries connections or more than one database, you can use the OleDb provider from Microsoft to bridge to the IBMDASQL OLE DB provider, which is included with iSeries Access for Windows starting with V5R3M0. Another alternative is to use the Microsoft ODBC provider to bridge to the ODBC driver included with iSeries Access for Windows. You can also use the IBM DB2 for LUW provider described in Chapter 5, "IBM DB2 for LUW .NET provider" on page 177.

Your application can manually enlist in a distributed transaction by calling the `EnlistDistributedTransaction()` on the connection object, or you can perform automatic enlistment by specifying `Enlist=true` in the `ConnectionString`. Read more about distributed transactions in “Performing distributed transactions with the DB2 LUW provider” on page 212.

Isolation levels

When executing commands on the iSeries, each application (or activation group) may or may not be affected by changes made by other applications. *Isolation level* refers to how isolated one application is from changes made by another application. Within ADO.NET, isolation levels are used when you execute statements within a transaction (after calling `BeginTransaction` but before a `Commit` or `Rollback` is performed). For a description of the isolation levels supported by IBM DB2 UDB for iSeries, refer to the *DB2 Universal Database for iSeries SQL Reference*, which you can find in the iSeries Information Center at:

<http://www.iseries.ibm.com/infocenter>

Select **Database** → **Reference** → **SQL Reference** → **Concepts** → **Isolation level**.

The `IBM.Data.DB2.iSeries` provider enables you to specify a transaction’s isolation level using any of several methods:

- ▶ Set the `DefaultIsolationLevel` property in your `iDB2Connection`’s `ConnectionString`. The isolation level you specify is used on all transactions for that connection unless you pass a different isolation level when you call the `BeginTransaction` method. Note that the `DefaultIsolationLevel` takes effect only when you are in a transaction. Statements executed outside of a transaction boundary always run with an isolation level of no commit or `*NONE`. See “`DefaultIsolationLevel`” on page 59 for a discussion of `DefaultIsolationLevel`.
- ▶ Include the desired isolation level when you call the `iDB2Connection` object’s `BeginTransaction` method.
- ▶ If you do not set the `DefaultIsolationLevel` property in your `ConnectionString`, and if you do not pass an isolation level when you call the `BeginTransaction` method, then the connection’s default isolation level of `ReadUncommitted` is used.

See Table 4-1 on page 59 for a list of the isolation levels supported by the provider, and the corresponding `System.Data.IsolationLevel` values you use to set the isolation level.

By default, the `IBM.Data.DB2.iSeries` provider runs with an isolation level of No commit or `*NONE`. This means that all changes made to the database are permanent; no explicit commit or rollback can be performed.

Autocommit

When you are not in transaction mode (that is, before you call `BeginTransaction()` on the connection object, and after you call `Commit()` or `Rollback()` on the transaction object), the `IBM.Data.DB2.iSeries` provider runs with an isolation level of No commit or `*NONE`. This means that all changes made to the database are permanent; no explicit commit or rollback can be performed. `*NONE` enables you to update non-journaled files. The provider does not currently allow you to specify an isolation level to be used while in autocommit mode.

Transactions and stored procedures

When you create a stored procedure on the iSeries, the SQL precompiler has an option that enables you to specify the isolation level the stored procedure runs under (`COMMIT` precompiler option). In some cases, that stored procedure isolation level is not used when the stored procedure is called from the .NET provider.

When using IBM.Data.DB2.iSeries with an iSeries host version earlier than V5R3M0, the stored procedure's isolation level is not used; instead, the isolation level in use by the .NET application is used. For example, if your .NET application has not started a transaction (using `BeginTransaction`), then the application is running under an isolation level of No commit or *NONE. Even if the stored procedure is compiled with an isolation level (using the COMMIT precompiler option), that isolation level may not be honored.

When using IBM.Data.DB2.iSeries with an iSeries host version of at least V5R3M0, the stored procedure's isolation level is used. V5R3M0 does not have the same restriction as older iSeries hosts, because the underlying commitment control interface used by the provider has been enhanced.

4.6.4 Calling a program by wrapping it in a stored procedure

Because the IBM.Data.DB2.iSeries provider is a SQL-only provider, you cannot directly call commands or programs that reside on the iSeries server. You can, however, use the SQL `CREATE PROCEDURE` statement to create a stored procedure that “wraps” your program. When you create the procedure, you define the input and output parameters and call the program from within the stored procedure. Then, you can call the stored procedure from your .NET application just as you can call any stored procedure.

This method can be used to call any program, including programs that use output parameters.

4.6.5 Calling a program or CL command using QCMDEXC

Throughout most of this chapter, our examples use only SQL statements. Sometimes you may want to call a program or CL command on the iSeries using the IBM.Data.DB2.iSeries provider. You can do this by wrapping the program in a stored procedure call as described in 4.6.4, “Calling a program by wrapping it in a stored procedure” on page 120.

You can also call a program or CL command by using the QCMDEXC API, located in the QSYS schema. QCMDEXC takes two input parameters:

- ▶ A string containing the text of the command or program you want to execute. If the string contains a single quote character ('), then you must delimit the quote with an extra quote as we show in Example 4-94 on page 121.
- ▶ A (packed) DECIMAL(15, 5) value containing the length of your command text.

You cannot use parameter markers with QCMDEXC, and you cannot receive any output parameters back from the call. In this section, we include some helpful tips for using QCMDEXC. You can read more about QCMDEXC in the iSeries Information Center at:

<http://www.iseries.ibm.com/infocenter>

Select **Programming** → **APIs** → **API finder**. Under **Find by name**, type QCMDEXC and click **GO**.

A general method for invoking QCMDEXC

The QCMDEXC parameters must be in a particular format, so you can use the `CallPgm` method shown in Example 4-94 on page 121 to invoke a command or program through QCMDEXC. It formats the command text and length parameters, then calls QCMDEXC. If the call completes without an error, `CallPgm` returns a success value of true. Otherwise, it returns a failure value of false. Copy the following code into a C# Console application.

Example 4-94 CallPgm method used to invoke QCMDEXC

```
// -----  
// This method runs a command on the iSeries using QCMDEXC.  
//  
// cmdtext is the command or program you want to call.  
// cn is an open iDB2Connection the command will be run on.  
// If the command runs without error, this method returns true.  
// -----  
static bool CallPgm(String cmdtext, iDB2Connection cn)  
{  
    bool rc = true;  
  
    // Construct a string which contains the call to QCMDEXC.  
    // Because QCMDEXC uses single quote characters, we must  
    // delimit single quote characters in the command text  
    // with an extra single quote.  
    String pgmParm = "CALL QSYS.QCMDEXC('"  
        + cmdtext.Replace("'", "'')"  
        + "', "  
        + cmdtext.Length.ToString("0000000000.00000")  
        + ")";  
  
    // Create a command to execute the program or command.  
    iDB2Command cmd = new iDB2Command(pgmParm, cn);  
  
    try  
    {  
        cmd.ExecuteNonQuery();  
    }  
    catch (Exception)  
    {  
        rc = false;  
    }  
  
    // Dispose the command since we're done with it.  
    cmd.Dispose();  
  
    // Return the success or failure of the call.  
    return rc;  
}
```

Example scenarios using QCMDEXC

The following examples show how to call the CallPgm example shown in the previous section.

Calling a program with a string input parameter

For this example, create a CL program on the iSeries called MYCLPGM using the CL source code shown in Example 4-95. Place the program into your sampledb schema. This program sends a message to the user ID *MYUSERID*. (Substitute your own iSeries user ID for *MYUSERID* in this example.)

Example 4-95 Creating the MYCLPGM sample program that uses a string input parameter

```
PGM PARM(&STRING1)  
  DCL VAR(&STRING1) TYPE(*CHAR) LEN(50)  
  
      SNDMSG MSG(&STRING1) TOUSR(MYUSERID)  
ENDPGM
```

To call the MYCLPGM program through QCMDEXC, we invoke the CallPgm method we show in Example 4-94 on page 121. Example 4-96 shows how you can call the program, and pass in different parameters. In this example, we assume you created your MYCLPGM program into a schema called MYSCHEMA.

- ▶ In the first call, the string parameter does not contain any embedded quote characters.
- ▶ In the second call, the string parameter contains an embedded single quote character. Because QCMDEXC surrounds the command text with a single quote character, we must delimit the single quote in our command text with an additional single quote as shown.
- ▶ In the third call, the string parameter contains an embedded double quote character. Because C# uses the double quote character to delimit strings, we must prefix the double quote character with an escape character ('\') as shown.

Example 4-96 Calling our sample program through QCMDEXC using our CallPgm method

```
// Create and open a connection to the iSeries.
iDB2Connection cn = new iDB2Connection("DataSource=myiseries;");
cn.Open();

// Call the MYCLPGM program three times.
// If the call fails, we don't continue.
String cmdtext = "call sampledb/myclpgm parm('this string contains no quote characters')";
bool rc = CallPgm(cmdtext, cn);

if (rc == true)
{
    cmdtext = "call sampledb/myclpgm parm('this string contains a single quote character(')')");
    rc = CallPgm(cmdtext, cn);
}

if (rc == true)
{
    cmdtext = "call sampledb/myclpgm parm('this string contains a double quote character(\")')");
    rc = CallPgm(cmdtext, cn);
}

if (rc == true)
    Console.WriteLine("The calls completed successfully.");
else
    Console.WriteLine("One of the calls failed.");

// Close the connection since we're done using it.
cn.Close();
```

Note: You can also use the SBMJOB CL command to run a program as a batch job. Extending our previous example, you could use the following to submit the call to MYCLPGM as a batch job:

```
rc = CallPgm("sbmjob cmd(call sampledb/myclpgm parm('this string was sent via SBMJOB'))", cn);
```

Accessing physical files with multiple members

Developers who are familiar with iSeries DDS programming can use DDS to create a database file that contains more than one member. SQL restricts you to using only the first member of a database file. You can access a member other than the first by using one of the following methods:

- ▶ Use an ALIAS to access the desired member.
- ▶ Use OVRDBF to override the multiple-member file.

Using an ALIAS is preferred for accessing multiple-member files. An ALIAS performs better than an override, and because it is a permanent object, it only has to be created once.

To illustrate use of a multiple-member file, assume that you have a file in schema MYSCHEMA called MYFILE, which contains two members, MEMBER1 and MEMBER2.

Using an ALIAS to access a multi-member file

To use an ALIAS to access a member other than the first member of a file, first create the alias, and then use that ALIAS instead of the file name. Example 4-97 shows how to create an alias to the second member of file MYFILE, then use that alias to reference the member.

Example 4-97 Accessing a multi-member file using an ALIAS

```
iDB2Command cmd = new iDB2Command("create alias myschema.fileMbr2 for
myschema.myfile(member2)", cn);
cmd.ExecuteNonQuery();

// Now access the second member using the alias we just created
cmd.CommandText = "select * from myschema.fileMbr2";
iDB2DataReader dr = cmd.ExecuteReader();

// Etc.
```

Using OVRDBF to access a multi-member file

You can use the OVRDBF command to temporarily override a multiple-member database file as shown in Example 4-98. Using the CallPgm example we create in “A general method for invoking QCMDEXC” on page 120, we override the file so that SQL will reference MEMBER2 instead of the first member in the file. After calling OVRDBF as we show in this example, SQL statements that reference the file MYSCHEMA.MYFILE use the second member, MEMBER2.

Example 4-98 Using OVRDBF to access the second member of a multi-member file

```
rc = CallPgm("OVRDBF FILE(MYFILE) TOFILE(MYSCHEMA/MYFILE) MBR(MEMBER2) OVRSCOPE(*JOB)",
cn);

// Now a SELECT * FROM MYSCHEMA.MYFILE will access member MEMBER2
cmd.CommandText = "select * from myschema.myfile";
iDB2DataReader dr = cmd.ExecuteReader();

// Etc.
```

Processing CL commands that produce an OUTFILE

Many CL commands provide an option to store the command output into an *outfile*, which is a database file that can be queried using a SQL select statement, just like any other table. In Example 4-99 on page 124, we execute a DSPUSRPRF command through QCMDEXC using the CallPgm method we created in Example 4-94 on page 121. After running this command, we execute a SELECT statement to select all of the user profiles from the table (outfile) that was created by DSPUSRPRF. We use a DataReader to read information from the table.

Example 4-99 Processing a CL command that produces an OUTFILE

```
// Create and open a connection to the iSeries.
iDB2Connection cn = new
iDB2Connection("DataSource=myiseries;DefaultCollection=samledb;");
cn.Open();

// Call the CallPgm method to execute the DSPUSRPRF command
// and tell it to put the output into an OUTFILE.
bool success = CallPgm("DSPUSRPRF USRPRF(*ALL) OUTPUT(*OUTFILE)
OUTFILE(SAMPLEDB/USRPRFINFO", cn);

// If the call succeeded, open a DataReader to read a list of
// all the user profiles in the outfile.
if (success == true)
{
    iDB2Command cmd = new iDB2Command("SELECT UPUPRF FROM USRPRFINFO", cn);
    iDB2DataReader dr = cmd.ExecuteReader();
    while (dr.Read())
    {
        Console.WriteLine("User profile: " + dr.GetString(0));
    }

    // Close the DataReader since we're done using it.
    dr.Close();

    // Dispose the command since we no longer need it.
    cmd.Dispose();
}

// Close the connection since we're done using it.
cn.Close();
```

Processing CL commands that produce a spool file

Some CL commands do not produce an OUTFILE like the command in Example 4-99 does. However, many allow you to spool the output. In this section, we show how to process the results of a command that produces a spool file. First, we create a program on the iSeries host that copies a spool file into a database file. Then we show an example that calls the DSPAUTUSR to generate a spool file, calls the program to copy the spool file into a database file, and reads from the database file to get the user information.

Follow these steps to process a spool file programmatically:

1. Create a CL program on the iSeries server called SAMPLEDB/SAVESPLF. This program uses the QSPRILSP API to determine the name and job information of the most recently created spool file for our job. The program takes a schema, library, and member name as input and copies the most recent spool file into that file. It then deletes the spool file.
 - To create the program, create a source physical file to hold the source code:
CRTSRCPF FILE(SAMPLEDB/SRCPF)
 - Add a member to this file:
ADDPFM FILE(SAMPLEDB/SRCPF) MBR(SAVESPLF) SRCTYPE(CLP)
 - Edit the file:
STRSEU SRCFILE(SAMPLEDB/SRCPF) SRCMBR(SAVESPLF) TYPE(CLP) OPTION(2)
 - Add the code shown in Example 4-100 on page 125 to the file.

Example 4-100 Source code for SAMPLEDB/SAVESPLF program

```
PGM PARM(&LIB &FILE &MBR)

/* Input parameters */
DCL &LIB *CHAR 10
DCL &FILE *CHAR 10
DCL &MBR *CHAR 10

/* Parameters used when calling the QSPRILSP API */
DCL &RCVVAR *CHAR 70
DCL &RCVLNG *INT 4
DCL &FORMAT *CHAR 8
DCL &ERRCODE *CHAR 8

/* Parameters used when calling CPYSPLF */
DCL &SPLFNAME *CHAR 10
DCL &SPLFNBR *INT 4
DCL &JOBNAME *CHAR 10
DCL &USERNAME *CHAR 10
DCL &JOBNBR *CHAR 6
DCL &CRTDATE *CHAR 8
DCL &CRTTIME *CHAR 6
DCL &JOBSYSNAM *CHAR 8

/* Variables used for the RTVJOBA CL command */
DCL &DATFMT *CHAR 4

/* Work vars */
DCL &UPYEAR *CHAR 2
DCL &LOWYEAR *CHAR 2
DCL &MONTH *CHAR 2
DCL &DAY *CHAR 2
DCL &HOUR *CHAR 2
DCL &MINUTE *CHAR 2
DCL &SECOND *CHAR 2

/* Initialize the input parameters */
CHGVAR &RCVLNG VALUE(70)
CHGVAR &FORMAT VALUE('SPRL0100')
CHGVAR %BIN(&ERRCODE 1 4) VALUE(0)
CHGVAR %BIN(&ERRCODE 5 4) VALUE(0)

/* Retrieve the exact identity of the most recent spool file */
CALL QSPRILSP PARM(&RCVVAR +
                  &RCVLNG +
                  &FORMAT +
                  &ERRCODE)

/* Set up the parameters for cpysplf */
CHGVAR &SPLFNAME VALUE(%SST(&RCVVAR 9 10))
CHGVAR &JOBNAME VALUE(%SST(&RCVVAR 19 10))
CHGVAR &USERNAME VALUE(%SST(&RCVVAR 29 10))
CHGVAR &JOBNBR VALUE(%SST(&RCVVAR 39 6))
CHGVAR &SPLFNBR VALUE(%BIN(&RCVVAR 45 4))
CHGVAR &JOBSYSNAM VALUE(%SST(&RCVVAR 49 8))

/* Convert the date from the QSPRILSP format to temp variables */
IF COND(%SST(&RCVVAR 57 1) *EQ '0') THEN(CHGVAR &UPYEAR VALUE('19'))
ELSE (CHGVAR &UPYEAR VALUE('20'))
CHGVAR &LOWYEAR VALUE(%SST(&RCVVAR 58 2))
```

```

CHGVAR &MONTH VALUE(%SST(&RCVVAR 60 2))
CHGVAR &DAY VALUE(%SST(&RCVVAR 62 2))

/* Convert the time from the QSPRILSP format to temp variables */
CHGVAR &HOUR VALUE(%SST(&RCVVAR 64 2))
CHGVAR &MINUTE VALUE(%SST(&RCVVAR 66 2))
CHGVAR &SECOND VALUE(%SST(&RCVVAR 68 2))

/* Get the job date format */
RTVJOBA DATFMT(&DATFMT)

/* Format for all date formats except julian */
SELECT
WHEN (&DATFMT *EQ 'YMD') +
  THEN(CHGVAR &CRTDATE VALUE(&UPYEAR *CAT &LOWYEAR *CAT &MONTH *CAT &DAY))
WHEN (&DATFMT *EQ 'MDY') +
  THEN(CHGVAR &CRTDATE VALUE(&MONTH *CAT &DAY *CAT &UPYEAR *CAT &LOWYEAR))
WHEN (&DATFMT *EQ 'DMY') +
  THEN(CHGVAR &CRTDATE VALUE(&DAY *CAT &MONTH *CAT &UPYEAR *CAT &LOWYEAR))
OTHERWISE
ENDSELECT

CHGVAR &CRTTIME VALUE(%SST(&RCVVAR 65 6))

/* Copy the spooled file to the database file and library passed to us */
CPYSPLF FILE(&SPLFNAME) TOFILE(&LIB/&FILE) JOB(&JOBNNBR/&USERNAME/&JOBNAME) +
  SPLNNBR(&SPLFNNBR) JOBSYSNAME(&JOBSYSNAM) CRTDATE(&CRTDATE &CRTTIME) +
  TOMBR(&MBR) MBROPT(*REPLACE) CTLCHAR(*NONE)

/* Delete the spool file */
DLTSPLF FILE(&SPLFNAME) JOB(&JOBNNBR/&USERNAME/&JOBNAME) +
  SPLNNBR(&SPLFNNBR) JOBSYSNAME(&JOBSYSNAM) CRTDATE(&CRTDATE &CRTTIME)

ENDPGM

```

- Save your source file, and then compile it:

```

CRTCLPGM PGM(SAMPLEDB/SAVESPLF) SRCFILE(SAMPLEDB/SRCPF) SRCMBR(SAVESPLF)

```

2. Now we write some code that can use the SAVESPLF program. In Example 4-101, we invoke the DSPAUTUSR command through QCMDXC using the CallPgm method we created earlier (see Example 4-94 on page 121). The DSPAUTUSR command produces a spool file called QPAUTUSR. Next, we call the SAVESPLF program we created in step 1 on page 124 to copy the spool file into a database file. Finally, we open a DataReader to read the user information from the database file.

Example 4-101 Processing a CL command that produces a spool file

```

// Create and open a connection to the iSeries.
iDB2Connection cn = new
iDB2Connection("DataSource=myiseries;DefaultCollection=sampled;");
cn.Open();

// Call the CallPgm method to execute the DSPAUTUSR command
// and tell it to spool the output.
bool success = CallPgm("DSPAUTUSR OUTPUT(*PRINT)", cn);

// If the call succeeded, create a temp file and call the
// SAVESPLF program, which will copy the spool file into
// our database file.
if (success == true)

```

```

{
    // Create the file. We don't care if it already exists.
    CallPgm("CRTPF FILE(SAMPLEDB/DSPAUTUSR) RCDLEN(132)", cn);

    // Clear the file (in case it already has data in it).
    CallPgm("CLRPFM FILE(SAMPLEDB/DSPAUTUSR) MBR(*FIRST)", cn);

    // Call the SAVESPLF program.
    // This program will copy the spool file we just
    // created into the file we specify.
    success = CallPgm("CALL SAMPLEDB/SAVESPLF PARM(SAMPLEDB DSPAUTUSR *FIRST)", cn);
}

// If we get to here without any errors, then we should have a
// file in our SAMPLEDB schema called DSPAUTUSR, which contains
// a list of authorized users on the system.
if (success == true)
{
    // Open a DataReader to read a list of the authorized
    // users on our system.
    iDB2Command cmd = new iDB2Command("SELECT * FROM DSPAUTUSR", cn);
    iDB2DataReader dr = cmd.ExecuteReader();
    while (dr.Read())
    {
        // To make our example shorter, we'll only
        // look at the user profile part of the string,
        // which is the first 10 characters.
        String userProfile = dr.GetString(0);
        Console.WriteLine(userProfile.Substring(0, 10));
    }

    // Close the DataReader since we're done using it.
    dr.Close();

    // Dispose the command since we no longer need it.
    cmd.Dispose();
}

// Close the connection since we're done using it.
cn.Close();

```

4.6.6 Choosing between iDB2DataReader and iDB2DataAdapter

In this section we discuss some differences between DataReaders and DataAdapters, and offer information about when to use which data access method in your application. Some of these topics are also discussed in Chapter 3, “ADO .NET object hierarchy” on page 17.

Differences between a DataReader and a DataAdapter

In the ADO.NET object model, the DataReader is used to read data from a database. The DataReader requires an open connection, and it reads a single row at a time, in a forward-only, read-only manner. The DataReader is similar to the OLE DB notion of a forward-only, read-only cursor. One advantage of a DataReader is that you work with only a single row of data at a time. This is especially important when your data set is large.

The DataAdapter, on the other hand, can be used to both read data from and write data to a database. The data from the database is cached in an ADO.NET object called a DataSet. While a DataReader requires an open connection to the database, the DataAdapter only

needs to keep the connection open while it is actively reading or writing to the database. Because the DataAdapter keeps a cache of all your data in the DataSet object, you can access all the rows of your data at the same time.

Many of the examples in the MSDN library (<http://msdn.microsoft.com/library/>) use DataAdapters because they are often the easiest way to read and write data. However, there are times when a DataReader is a better choice. When deciding between a DataReader and a DataAdapter, you must consider several variables, including the size and type of your data, and how you plan to use it.

When to use a DataReader

While a DataAdapter is often the easiest way to read and write data to and from your iSeries server, here are some cases when a DataReader might be a better fit for your application:

- ▶ If you are working with a large amount of data, such as a table containing a large number of records. Because the DataAdapter keeps a copy of your entire data set in memory, a selection from a very large table could cause too much memory to be consumed by your .NET application to make it feasible to work with. Using a DataReader enables you to process a single row at a time, thus your application does not consume as much memory.
- ▶ If you are working with a table that contains large objects (LOBs). LOBs are normally used to hold large amounts of data. If you try to use a DataAdapter with a table containing LOBs, all of the data in the table is read into the DataSet at once, and thus into your application's memory. This is true whether or not you read the LOB data using inline LOBs or LOB locators (see "MaximumInlineLobSize" on page 58). As we mention in the previous item, using a DataReader enables you to process a single row at a time and gives you control over whether or when the LOB data is actually read by your application.

Note: LOB support was added to the IBM.Data.DB2.iSeries provider in V5R3M0 service pack SI15176.

For more information about using LOBs with the IBM.Data.DB2.iSeries provider, see 4.7.2, "Using large objects (LOBs)" on page 132.

- ▶ If your data contains characters data tagged with CCSID 65535, but you want to read and write that data as strings. By default, character data tagged with CCSID 65535 is treated by the DataAdapter as binary data (an array of bytes). If you want to treat this type of data as a string, you must either use a DataReader, or take special action to use it with a DataAdapter. See "Reading non-binary data tagged with CCSID 65535" on page 91 for more information.
- ▶ If your data contains TIMESTAMP data, and you need the ability to read or write the full microsecond precision, you must either use a DataReader or take special action to cast the TIMESTAMP to a character string. (See "iDB2TimeStamp" on page 95.)
- ▶ If your data contains time or timestamp values that contain the special value of 24:00:00 for midnight. (See "TIME and TIMESTAMP special values" on page 96.)
- ▶ If your data contains decimal or numeric values that have a precision larger than the .NET Framework Decimal data type supports. (See "iDB2Decimal and iDB2Numeric" on page 97.)
- ▶ If you know that you do not need to read the entire query result at once. This could happen if you want to look through the query result and stop when you reach a certain record.
- ▶ If you know that you do not need to update the data. Because a DataReader is forward-only, read-only, it can improve your application's performance.

When to use a DataAdapter

As we discuss earlier, the DataAdapter is often the easiest way to move data back and forth between your application and the iSeries database. The following list includes some cases where a DataAdapter might be a better choice for your application than a DataReader:

- ▶ When your application may update the data. When used with a CommandBuilder, the DataAdapter makes updates relatively easy and painless. For more information about updating DataSets, see 4.7.3, “Updating DataSets” on page 136.
- ▶ When your application works with a relatively small amount of data, and the data does not contain certain data types. (See the previous section, “When to use a DataReader” on page 128.)
- ▶ When your application plans to use all of the data at once. Because the DataAdapter keeps a cache of your data in a DataSet, you can access all of the data easily. The DataReader, on the other hand, can only access data in a forward-only, read-only manner.

4.7 Advanced topics

This section covers some more-advanced topics affecting the IBM.Data.DB2.iSeries .NET provider.

4.7.1 Internationalization and support for multiple languages

In this section we discuss some of the features of the IBM.Data.DB2.iSeries provider that enable your applications to work regardless of your language or culture settings.

Support for multiple languages

The IBM.Data.DB2.iSeries provider is part of the iSeries Access for Windows product, which includes support for many different languages. In fact, you can install support for more than one language at a time. Read about this in the iSeries Access for Windows User's Guide. To display the User's Guide from the Windows desktop, select **Start** → **Programs** → **IBM iSeries Access for Windows** → **User's Guide**. In the User's Guide, click the **Contents** tab and expand **Using iSeries Access for Windows** → **Languages** → **Installing secondary languages**. For example, secondary language support can be useful if you have a Web page that supports more than one language.

At runtime, whenever the IBM.Data.DB2.iSeries provider displays any translatable text, the text displays in the language specified by your application's *Thread.CurrentCulture* setting, as long as you have installed the appropriate primary or secondary language. If the language is not installed, the text displays in a default language. An example of translatable text used by the provider is the Message property of an iDB2Exception object. The **cwbmptrc** provider trace utility also displays its help text using the CurrentUICulture setting (usually the default language setting for your PC). Example 4-102 shows a simple example that illustrates how this works. First, using the default culture setting, we force an exception to occur and display the exception Message property. The message is displayed in the default language. Next, we set the CurrentUICulture to a different language, and again force the exception to occur. This time, when we display the exception Message property, the message is displayed in the secondary language. Before running this example, you should install both a primary and a secondary language using iSeries Access for Windows. Change the code in the example to use the secondary language that you install.

Example 4-102 Using the CurrentUICulture to change your language setting

```
// Create a connection object.  
iDB2Connection cn = new iDB2Connection();
```

```

// Using the default culture, initialize the
// ConnectionString to an invalid value.
// We expect this to fail.
try
{
    cn.ConnectionString = "Oops, invalid connection string!";
}
catch (iDB2Exception e)
{
    Console.WriteLine("Exception: " + e.Message);
}

// Now, change the culture to "fr-FR".
CultureInfo currCulturex = new CultureInfo("fr-FR");
Thread.CurrentThread.CurrentUICulture = currCulturex;

// Initialize the ConnectionString to an invalid value again.
// This time, the exception message shows up in the
// new language.
try
{
    cn.ConnectionString = "Oops, invalid connection string!";
}
catch (iDB2Exception e)
{
    Console.WriteLine("Exception: " + e.Message);
}

```

Note: If you run this example and you do not have the secondary language installed, both of the exceptions display the Message property in the default language. Also, not all exceptions use translated text. For example, for the iDB2SQLException, the Message property is text received from the iSeries server.

Other culture-specific settings

Some cultures expect to work with dates, times, numbers, strings comparisons, and other items differently from the way other cultures use them. The .NET Framework uses the thread's *CurrentCulture* property to handle these culture-specific settings. The IBM.Data.DB2.iSeries provider uses the CurrentCulture setting when parsing decimal, numeric, date, time, and timestamp data. The CurrentCulture setting also affects the format of string data. To illustrate one of these differences, we compare the number format between en-US (United States English) and de-DE (Germany German). Example 4-103 displays a decimal value using two different cultures, to show how the data is returned using the current culture's default decimal separator character. Our example uses both Decimal and iDB2Decimal to show that the provider handles the different cultures seamlessly.

Example 4-103 Using the CurrentCulture to change how decimal data is displayed

```

// In this example, we display the value represented by:
// 392<decimal separator>02
//
// With "en-US", this number displays as:
// 392.02
// because the period is the decimal separator.
//
// With "de-DE", this number displays as:
// 392,02
// because the comma is the decimal separator.

```

```
// Construct a decimal and an iDB2Decimal
Decimal d = new Decimal(39202, 0, 0, false, 2);
iDB2Decimal idb2dec = new iDB2Decimal(d);

// Set the culture to "en-US" and display the
// decimal values.
CultureInfo currCulturex = new CultureInfo("en-US");
Thread.CurrentThread.CurrentCulture = currCulturex;
Console.WriteLine("en-US setting (Decimal): " + d.ToString());
Console.WriteLine("en-US setting (iDB2Decimal): " + idb2dec.ToString());

// Set the culture to "de-DE" and display the
// decimal values.
currCulturex = new CultureInfo("de-DE");
Thread.CurrentThread.CurrentCulture = currCulturex;
Console.WriteLine("de-DE setting (Decimal): " + d.ToString());
Console.WriteLine("de-DE setting (iDB2Decimal): " + idb2dec.ToString());
```

The `CurrentCulture` setting affects both how you write data (in this instance, the `ToString()` method) and how data is read by the provider. For example, if your `CurrentCulture` setting is `en-US` and you initialize an `iDB2Decimal` object with the string `111.111`, the provider treats this value as “one hundred eleven<decimal separator>one one one.” By comparison, if you initialize an `iDB2Decimal` object with the same string, and your `CurrentCulture` setting is `de-DE`, then the provider treats this value as “one hundred eleven thousand, one hundred and eleven,” with no decimal separator. Run the code in Example 4-104 to see how the two values are treated differently.

Example 4-104 Using the `CurrentCulture` to change how decimal data is handled by the provider

```
// Set the culture to "en-US" and construct a
// Decimal and an iDB2Decimal value.
CultureInfo currCulturex = new CultureInfo("en-US");
Thread.CurrentThread.CurrentCulture = currCulturex;
iDB2Decimal idb2decimalUS = new iDB2Decimal("111.111");
Console.WriteLine(idb2decimalUS.Value.ToString());

// Set the culture to "de-DE" and display the
// decimal values.
currCulturex = new CultureInfo("de-DE");
Thread.CurrentThread.CurrentCulture = currCulturex;
iDB2Decimal idb2decimalDE = new iDB2Decimal("111.111");
Console.WriteLine(idb2decimalDE.Value.ToString());

// Compare the two values and make sure they
// do not compare equal.
if (idb2decimalUS.CompareTo(idb2decimalDE) == 0)
    Console.WriteLine("The two values are equal. This should not happen.");
else
    Console.WriteLine("The two values are not equal. This is what we expect.");
```

Note: In “`iDB2Decimal` and `iDB2Numeric`” on page 97 and Example 4-81 on page 99, we discuss casting a decimal or numeric value to a string. When casting the decimal/numeric as in those examples, the decimal separator—decimal point (`.`) or comma (`,`)—is *always* returned as a period (`.`), regardless of your thread’s `CurrentCulture` setting.

Sort sequence

The IBM.Data.DB2.iSeries provider enables you to change the sort sequence used by the DB2 UDB for iSeries server job that processes requests on behalf of your application. This is done using by setting the SortSequence and related properties in your iDB2Connection object's ConnectionString. See "SortSequence" on page 56 for more about setting the sort sequence.

4.7.2 Using large objects (LOBs)

Support for large objects (LOBs) was added to the IBM.Data.DB2.iSeries provider with iSeries Access for Windows V5R3M0, service pack SI15176. There are three types of LOBs supported by DB2 UDB for iSeries:

- ▶ Binary large objects (BLOBs)
- ▶ Character large objects (CLOBs)
- ▶ Double-byte character large objects (DBCLOBs)

The IBM.Data.DB2.iSeries provider lets you work with these LOBs using the provider-specific data types iDB2Blob, iDB2Clob, and iDB2DbClob, or using the built-in .NET Framework data types. (See Table 4-5 on page 88 for information about the default mappings between LOB data types and .NET Framework data types.) The provider also includes a ConnectionString property called MaximumInlineLobSize, which enables you to control how LOB data is transferred between your iSeries server and your application. See "MaximumInlineLobSize" on page 58 for more information.

Keep a few things in mind when working with LOBs: By their very nature, LOBs are normally associated with large amounts of data, so we recommend not using a DataAdapter when your result data contains LOB fields. Because the DataAdapter reads *all* of your query result data into memory at once, your available PC memory can be used up quickly when the result data contains LOBs.

Instead, use a different method to read your LOB data. Using a DataReader gives you greater control over whether or when your LOB data is read. Alternatively, use ExecuteScalar to read a single LOB value from your table.

A LOB example

As a starting point for our LOB example, we use the running application that we created in 4.6.2, "A simple DataAdapter with CommandBuilder example" on page 110. That example reads data from the EMPLOYEE table in our SAMPLEDB schema. We expand the example now to add a PictureBox element to the window, and use that PictureBox to display a picture of the selected employee. The employee picture is a BLOB field contained in the EMP_PHOTO table in the SAMPLEDB schema.

Starting from the DataAdapter example, add a PictureBox to the window. Begin by making the window larger by pulling the window to the right, then drag a PictureBox from the Toolbox to this expanded area of your window. Your window should look similar to Figure 4-28 on page 133, with the PictureBox on the right.

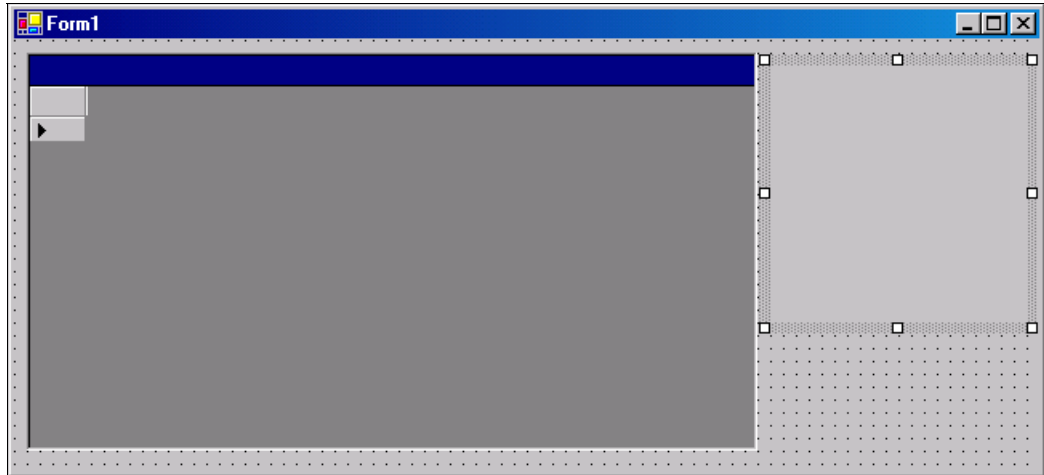


Figure 4-28 Add a PictureBox to your window

Double-click the window to display the `Form1_Load` method. Add this code to `Form1_Load` to register an event handler to handle the `CurrentCellChanged` event of the `DataGrid`:

```
dataGrid1.CurrentCellChanged += new EventHandler(OnCurrentCellChanged);
```

Scroll up nearly to the top of your source file, where variables are defined. Add the following code, which declares and initializes a variable called `currentRow`. This variable is used to keep track of which row in the `DataGrid` is selected:

```
private int currentRow = -1;
```

Scroll to the bottom of the source file and add the new `OnCurrentCellChanged` method after the `Form1_Load` method (Example 4-105). When you run your program, this method gets control each time you click a different cell in the `DataGrid`. `OnCurrentCellChanged` determines whether you have moved the focus to a new row within the `DataGrid`. If so, then it reads the `EMPNO` field from the `DataGrid` and uses that `EMPNO` field to select a single `BLOB` field from the `EMP_PHOTO` table using `ExecuteScalar`. If the `EMP_PHOTO` has a picture of the selected employee, that picture is displayed in the `PictureBox`.

Example 4-105 Add the `OnCurrentCellChanged` method to read the LOB field

```
protected void OnCurrentCellChanged(object sender, System.EventArgs e)
{
    // Find out which cell is now active
    int newRow = dataGrid1.CurrentCell.RowNumber;

    // If a new row has been activated, then update currentRow
    // and read the employee picture from EMP_PHOTO, if a picture exists.
    if (newRow != currentRow)
    {
        currentRow = newRow;

        if (dataGrid1[currentRow, 0] != System.DBNull.Value)
        {
            // Get the employee number (EMPNO) from the DataGrid.
            // The EMPNO is in column 0 of the DataGrid.
            String employeeNumber = (String)dataGrid1[currentRow, 0];

            // Create a command to read a record from the EMP_PHOTO table.
            iDB2Connection1.Open();
            iDB2Command cmd = iDB2Connection1.CreateCommand();
```

```

        cmd.CommandText = "select picture from emp_photo where empno=@empno and
photo_format='bitmap'";

        // Derive the parameter information
        cmd.DeriveParameters();

        // Select the record for the employee
        cmd.Parameters["@empno"].Value = employeeNumber;

        // Execute the command, and read the result into a byte array.
        Byte[] b = (Byte[])cmd.ExecuteScalar();
        iDB2Connection1.Close();

        if (b == null)
        {
            // If the byte array is empty, display an
            // empty picture box.
            pictureBox1.Height = 0;
            pictureBox1.Width = 0;
            pictureBox1.Show();
        }
        else
        {
            // If the byte array is not empty, put the byte array
            // into a memory image, and then use that to create an
            // Image. Then, display the image in our picture box.
            System.IO.MemoryStream ms = new System.IO.MemoryStream(b);
            System.Drawing.Image i = System.Drawing.Image.FromStream(ms);
            pictureBox1.Height = i.Height;
            pictureBox1.Width = i.Width;
            pictureBox1.Image = i;
            pictureBox1.Show();
        }
    }
}

```

After making these changes to your test case, save the application (**File** → **Save All**) and run it. Just as before, the contents of the EMPLOYEE table are displayed. This time, as you click on a grid element that has an employee with a record in the EMP_PHOTO table, that employee's picture is displayed in the PictureBox on the right side of your window.

Note: You may need to experiment with your window size and format to ensure that the employee picture is displayed without clipping. Most of the employees in the EMPLOYEE table do not have a picture in the EMP_PHOTO table, so you may need to scroll through several rows before you see a picture displayed.

Figure 4-29 on page 135 shows an example of the window with an employee photo displayed.

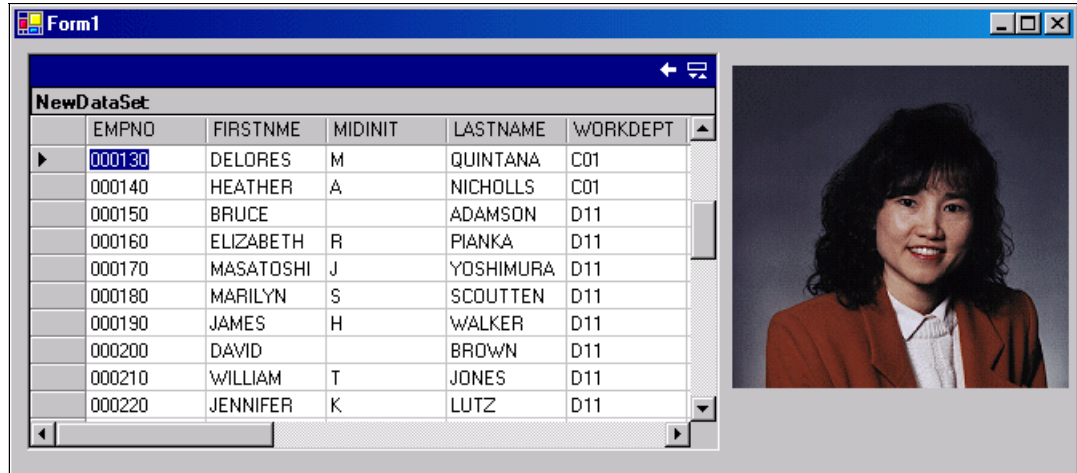


Figure 4-29 LOB example showing an employee photo next to the DataGrid

Other LOB techniques

Another LOB technique that helps reduce an application's resources uses a `DataReader` combined with the `GetBytes` or `GetChars` method to read the LOB data in smaller pieces. The code in Example 4-106 shows how to read the `RESUME` field from the `EMP_RESUME` table using `GetChars`. This example uses a Console application for simplicity. The `EMP_RESUME` table is in the `SAMPLEDB` schema (described in 1.4, "DB2 UDB for iSeries sample schema" on page 8). As the data is read in pieces, it is placed into an HTML file. After running Example 4-106, you should see a file called `resume.html` in the program's output directory. Double-click this file to see the employee's résumé information formatted using HTML.

Example 4-106 Using `GetChars` to read CLOB data in pieces

```
// Create and open a connection to the iSeries.
// To conserve memory, we set the MaximumInlineLobSize to zero
// so the provider will use LOB Locators to read the data.
String connectionString = "DataSource=myiseries; DefaultCollection=sampledb;
MaximumInlineLobSize=0;";
iDB2Connection cn = new iDB2Connection(connectionString);
cn.Open();

// Create a command which selects a record from the EMP_RESUME table.
iDB2Command cmdResume = cn.CreateCommand();
cmdResume.CommandText = "select resume from emp_resume where empno='000130' and
resume_format='html'";

// Open a DataReader to read the EMP_RESUME into a file called resume.html
iDB2DataReader drResume = cmdResume.ExecuteReader();
if (drResume.Read())
{
    if (drResume.IsDBNull(0) == false)
    {
        StreamWriter writer = new StreamWriter("resume.html", false);
        char[] chars = new char[100];
        long fieldOffset = 0;
        long charsRead = 0;

        while ((charsRead = drResume.GetChars(0, fieldOffset, chars, 0, chars.Length)) != 0)
        {
            writer.Write(chars);
            fieldOffset += charsRead;
        }
    }
}
```

```

        }
        writer.Close();
    }
}
drResume.Close();

// Dispose the command since we no longer need it.
cmdResume.Dispose();

// Close the connection.
cn.Close();

```

Note: Normally, you should not read data in only 100-character or 100-byte pieces, because if you have large amounts of data to read, this can have a negative impact on your network performance. This example reads the data in 100-character pieces just to illustrate how you *can* read LOB data in chunks.

Another way to limit the amount of LOB data an application uses is to limit the number of rows in your SELECT statement, so you only retrieve a small number of rows from the host. In the previous example, we limit the rows in our select statement using a WHERE clause to make sure we only get one row back.

4.7.3 Updating DataSets

In 4.6.2, “A simple DataAdapter with CommandBuilder example” on page 110, we show how to quickly and easily generate a DataGrid that contains data read from the EMPLOYEE table in the SAMPLEDB schema. In this section, we go a little further and show how to update data in the EMPLOYEE table. Before continuing this section, follow the steps in 4.6.2, “A simple DataAdapter with CommandBuilder example” on page 110 to create the basic application.

Note: If you already went through the LOB example shown in “A LOB example” on page 132, you can use that application as a starting point instead.

Writing code to update the DataSet

When you have the basic application working, go back to your window’s form design and follow these steps:

1. Make sure the Toolbox is displayed with the Windows Forms item expanded.
2. Click in your window (not on the DataGrid part of the window) to make it active.
3. Grab the bottom of the window and drag it down to enlarge it enough to hold two buttons.
4. From the Toolbox, drag a button to the bottom of your window. Right-click the button and select **Properties**. Change the Text from button1 to Read data. Change the (Name) from button1 to ReadButton. Press Enter.
5. From the Toolbox, drag another button to the bottom of your window. Right-click the button and select **Properties**. Change the Text from button1 to Update data. Change the (Name) from button1 to UpdateButton. Press Enter.

Your window should look similar to Figure 4-30 on page 137.

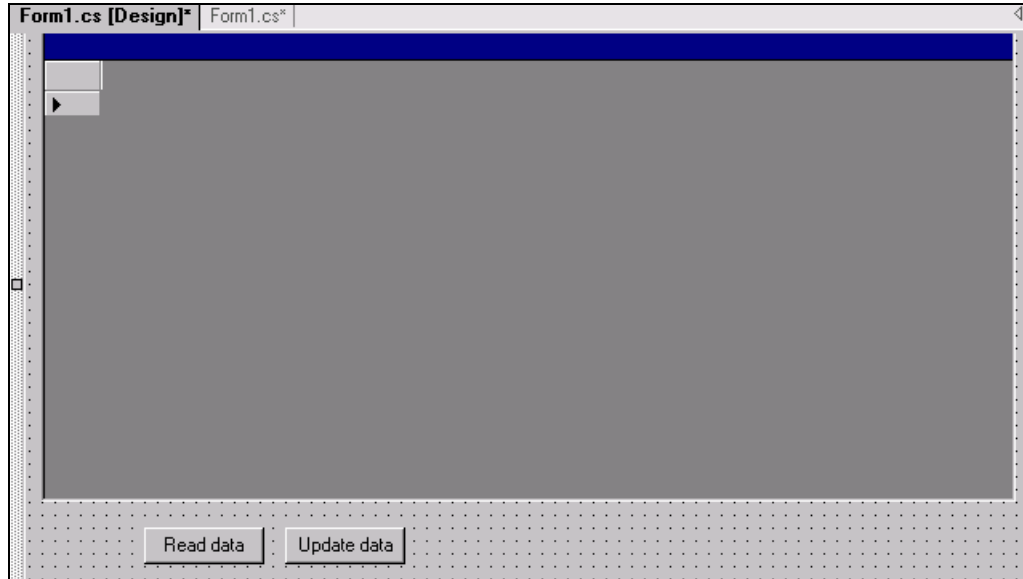


Figure 4-30 Add buttons to your window

To add code behind the buttons, double-click the **Read data** button. Visual Studio .NET creates a method called `ReadButton_Click`. Add the code in Figure 4-107 to this method.

Example 4-107 Add code to the `ReadButton_Click` method

```
dataSet1.Clear();
iDB2DataAdapter1.Fill(dataSet1);
```

Next, double-click the **Update data** button. Visual Studio .NET creates a method called `UpdateButton_Click`. Add the code shown in Figure 4-108 to the `UpdateButton_Click` method.

Example 4-108 Add code to the `UpdateButton_Click` method

```
iDB2Connection1.Open();
iDB2DataAdapter1.Update(dataSet1);
iDB2Connection1.Close();
```

Updating the `DataSet` requires an open connection to the `iSeries`, which explains adding the call to `Open` the connection in our `UpdateButton_Click` method. Because the window may stay up for a long time, we do not want the connection to stay active unless we are actively updating data, so we add the call to `Close` the connection after the update is performed.

Save your application (**File** → **Save All**). It is ready to run.

Running the updatable `DataSet` example

When you are ready, run the sample application. As in previous examples, a window is displayed that shows the data read from the `EMPLOYEE` table.

- To insert a new row, scroll to the bottom of the window to where the asterisk (`*`) is displayed. Position the cursor in the `EMPNO` column of the row containing the `*`. Fill in the information for a new row of data and press `Enter` to add the new row to the `DataSet`. The data is not sent to the `iSeries` host until the `Update` button is clicked. You can see this if you insert a new row but then click the `Read data` button before the `Update data` button. Because you reread the data before updating the `iSeries` table with the new data, the data you added to the `DataSet` is lost.

Note: The EMPLOYEE table requires the EMPNO to have a unique value for each employee, so use a value here that is not already in the table, such as 300010. Also, the WORKDEPT column must contain a department that exists in the DEPARTMENT table, so use a department that already exists, such as A00.

- ▶ To update an existing row, place the cursor on the data you want to modify and change it. The data is updated in the DataSet, but the updated data is not sent to the iSeries host until the Update button is clicked.
- ▶ To delete an existing row, click in the gray area to the left of the row you want to delete and click the **Delete** key. The row is deleted from the DataSet, but the data is not deleted from the iSeries host until the Update button is clicked.

You can continue adding, updating, and deleting data using the DataTable. When you are through making changes, click the **Update data** button. This sends all of the inserts, updates, and deletes to the iSeries host, and the EMPLOYEE table is updated to reflect the modified DataTable.

Updating a DataSet with optimistic concurrency

In our updatable DataSet example, we use a CommandBuilder to generate the Insert, Update, and Delete commands that are used when our DataAdapter's Update() method is called. The CommandBuilder generates these commands using an update model called *optimistic concurrency*. You can read more about concurrency at the MSDN Library Web site:

<http://msdn.microsoft.com/library/>

Select **.NET Development** → **.NET Framework SDK** → **.NET Framework** → **Programming with the .NET Framework** → **Accessing Data with ADO.NET** → **Sample ADO.NET Scenarios** → **Optimistic Concurrency**.

With optimistic concurrency, the provider is “optimistic” that “concurrent” changes to the database do not occur while the application is using the table. In simple terms, this means that the provider assumes that nobody else is using the table at the same time as the application. In a multi-user environment, two or more users or applications may want to update the same table. Optimistic concurrency provides a way to ensure that data updates are not made unless the row on the iSeries is the same as when it was read by the application. The provider does not lock any rows of data; rather, when an update is made, the provider checks the current data in the row against the data originally read from the row. If the data is the same, the provider assumes that it is safe to update the row. If the data is not the same, it means that another user or application modified the data after we read it but before we had a chance to update it, and a *concurrency violation* results.

Example of concurrency violation

As an example, run the updatable DataSet sample that we create earlier in this section, and while the application is still running, run another version of the same program. For the sake of our discussion, we call these programs APP1 and APP2 (even though they are two copies of the same application). Both APP1 and APP2 show the data from the EMPLOYEE table. Next, on the APP1 window, modify one of the existing rows as we just showed in “Running the updatable DataSet example.” Click **Update data** to send the updated row to the iSeries host. Go to the APP2 window, which still shows the original value that was read when you started the program, not the update that was just made in APP1. In the APP2 window, scroll to the row that you just updated from APP1, make an update to the row, and click **Update data**. A concurrency violation results in the application receiving a DBConcurrencyException. The update attempted by APP2 failed because the data in the row it tried to update was changed (by APP1) after APP2 read the data.

Using the RowUpdatedEvent to catch concurrency violations

A robust application has to handle these concurrency violations. The MSDN library Web site section entitled Optimistic Concurrency, which we referenced earlier in this section, shows how to use the RowUpdatedEvent to test for and handle concurrency violations. Our next example shows how to use the RowUpdatedEvent to handle the concurrency violation and continue, instead of terminating your application.

Starting with the updatable DataSet example that we used in “Writing code to update the DataSet” on page 136, add the following code to the bottom of your Form1_Load method:

```
iDB2DataAdapter1.RowUpdated += new iDB2RowUpdatedEventHandler(OnRowUpdated);
```

Next, add the new method shown in Figure 4-109 to your program.

Example 4-109 OnRowUpdated event handler

```
protected static void OnRowUpdated(object sender, iDB2RowUpdatedEventArgs args)
{
    if (args.RecordsAffected == 0)
    {
        args.Row.RowError = "Optimistic Concurrency Violation. Update not performed.";
        args.Status = UpdateStatus.SkipCurrentRow;
    }
}
```

The OnRowUpdated event handler runs whenever a row is updated. If it detects a concurrency violation (indicated by the RecordsAffected returning as 0), it sets the row's status to indicate an error.

After you add code to handle the concurrency violation, save your program (**File** → **SaveAll**) and run two versions of it, as in “Example of concurrency violation” on page 138. This time, because we handle the concurrency violation, instead of seeing an error pop up when you update APP2, you instead see a red exclamation point (!), indicating an error. The application continues. Place the mouse pointer over the exclamation point to see the error we set in the OnRowUpdated event handler routine:

```
Optimistic Concurrency Violation encountered. Update not performed.
```

4.7.4 Using iDB2CommandBuilder

In 4.7.3, “Updating DataSets” on page 136, we show how to update a DataSet using a CommandBuilder and a DataAdapter. The purpose of the CommandBuilder is to generate the Insert, Update, and Delete statements that are used by the DataAdapter. Using a CommandBuilder can prevent having to write a lot of code, so we encourage its use whenever possible. To use a CommandBuilder with IBM.Data.DB2.iSeries, follow these rules:

1. The CommandBuilder only works with SELECT statements; it cannot build statements when your SelectCommand is a stored procedure call.
2. The CommandBuilder works best with simple SELECT statements. This is especially true when working with older iSeries hosts. Use the following guidelines for your SELECT statements:
 - Simple statements produce the best results; for example:

```
SELECT * FROM SAMPLEDB.EMPLOYEE
```
 - Fully qualify the table name with its schema; for example:

```
SAMPLEDB.EMPLOYEE
```

- Selection fields are allowed, but must be specified in simple format. Only columns specified in the query table should be used; for example:

```
SELECT EMPNO, FIRSTNME, LASTNAME FROM SAMPLEDB.EMPLOYEE
```

- Derived fields or constants in the selection criteria are discouraged because they may produce unpredictable results or an exception; for example:

```
SELECT EMPNO, LASTNAME, LENGTH(LASTNAME) FROM SAMPLEDB.EMPLOYEE
```

3. All of the columns that are returned by your SELECT statement must be from the same base table and base schema. The following statement would cause the CommandBuilder to fail because the columns are from two different tables:

```
SELECT EMPNO, DEPTNO FROM SAMPLEDB.EMPLOYEE, SAMPLEDB.DEPARTMENT
```

4. In order for the CommandBuilder to generate Update and Delete commands, the columns that are returned by the SELECT statement must include a unique or primary key. In 4.7.3, “Updating DataSets” on page 136, our example used the EMPLOYEE table, whose EMPNO field constitutes a primary key. If you run that same example using the IN_TRAY table instead of the EMPLOYEE table, your attempts to update or delete a row would fail because the IN_TRAY table does not have a primary or unique key.

Note: Primary keys are recommended over unique keys because they do not allow nullable key values, so you cannot end up with multiple rows in your table with duplicate null key fields.

5. If the unique or primary key spans multiple columns, your SELECT statement must include all of the key columns, either using SELECT * or by naming each key column in the SELECT statement. If the example in 4.7.3, “Updating DataSets” on page 136 is changed to use the following SELECT statement, attempts to update or delete a row would fail because you did not include the ACSTDAT field in your SELECT statement (the key comprises the PROJNO, ACTNO, and ACSTDAT fields):

```
SELECT PROJNO, ACTNO FROM SAMPLEDB.PROJECT
```

If you cannot use the CommandBuilder for one of these reasons, you must build your own Insert, Update, and Delete statements.

Nullable columns: performance considerations

When the CommandBuilder generates Update and Delete statements, it uses optimistic concurrency to make the updates. (See “Updating a DataSet with optimistic concurrency” on page 138.) With optimistic concurrency, the original values of each column in the table to be updated are checked against the current value in each column. When you use nullable columns, an extra check must be made for each nullable column because the SQL standard requires a check for IS NULL before checking the column for a non-null value. This makes the Update or Delete statement more complex.

When you insert a row using the InsertCommand generated by the CommandBuilder, you must make sure not to send any null data to a column that is not nullable and has no default value. For example, if you run our updatable DataSet example (see “Updating a DataSet with optimistic concurrency” on page 138), and try to insert a new row but leave the FIRSTNME column null, you get an SQL exception when you click **Update**. The SQL exception says that null values are not allowed in the column.

Make sure the most recent fixes have been applied

If you run into problems using the CommandBuilder and they are not related to the previously discussed items, be sure that you are running with the most recent iSeries Access for

Windows service pack. See 4.3.1, “PC setup” on page 39 for information about applying fixes to iSeries Access for Windows. Also, make sure that the most recent fixes have been applied to your iSeries host. See 4.3.2, “Host setup” on page 40 for information about host fixes.

4.7.5 Using DataLinks

The DATALINK data type is not supported by the IBM.Data.DB2.iSeries provider as of this writing. However, you can use this type if you cast the datalink to a supported type (for example, a character string). In this section we include some examples of how you can use a table that contains a datalink field.

Selecting a DATALINK from a table

To read data from a table containing a DATALINK, use a `DataReader` with `GetString` to read the value. You must cast the DATALINK to a character string using one of the DATALINK functions, such as `DLURLCOMPLETE`, as shown in Example 4-110. For this example, we select the `DL_PICTURE` field from the `EMP_PHOTO` table in our `SAMPLEDB` schema. See 1.4, “DB2 UDB for iSeries sample schema” on page 8 to set up the `SAMPLEDB` schema.

Example 4-110 Selecting a DATALINK field from a table using DLURLCOMPLETE

```
// Create and open a connection to the iSeries.
iDB2Connection cn = new
iDB2Connection("DataSource=myiseries;DefaultCollection=sampledb;");
cn.Open();

// Create a command which selects the DATALINK field
// from the EMP_PHOTO table.
iDB2Command cmd = new iDB2Command("SELECT DLURLCOMPLETE(DL_PICTURE) FROM EMP_PHOTO", cn);

// Execute the command and get a DataReader in return.
iDB2DataReader dr = cmd.ExecuteReader();

// Read the Datalink field into a String.
while (dr.Read())
{
    if (dr.IsDBNull(0) == false)
        Console.WriteLine(dr.GetString(0));
}

// Close the DataReader since we're done with it.
dr.Close();

// Dispose the command since we're done with it.
cmd.Dispose();

// Close the connection once more.
cn.Close();
```

Note: You can also use a `DataAdapter` with a DATALINK if you cast the DATALINK to a character string in your `Select` statement as shown in Example 4-110. When using a DATALINK with a `DataAdapter`, you cannot use the `CommandBuilder`. Instead, you must build your own `Insert`, `Update`, and `Delete` statements.

Inserting, updating, or deleting a DATALINK field in a table

To insert, update, or delete a datalink field, you must use the `DLVALUE` function to turn a character string into a DATALINK, as shown in Example 4-111 on page 142. In this example,

we insert a row into the EMP_PHOTO table we used in our previous example. We use DLVALUE to cast the character string into a DATALINK value.

Example 4-111 Inserting a DATALINK field into a table using DLVALUE

```
// Create and open a connection to the iSeries.
iDB2Connection cn = new
iDB2Connection("DataSource=myiseries;DefaultCollection=sampledb;");
cn.Open();

// Create a command which inserts a row into the EMP_PHOTO table.
// The DATALINK field is inserted using the DLVALUE function.
iDB2Command cmd = new iDB2Command("INSERT INTO EMP_PHOTO VALUES('000010', 'bitmap', NULL,
'',
DLVALUE('HTTP://LP126AB.RCHLAND.IBM.COM/QIBM/ProdData/OS400/SQL/Samples/db200190.bmp')"),
cn);

// Execute the command.
cmd.ExecuteNonQuery();

// Dispose the command since we're done with it.
cmd.Dispose();

// Close the connection once more.
cn.Close();
```

Updating a DATALINK field using a parameter marker

If you want to use a variable value with a DATALINK, you can cast the parameter marker to a character string that is the same length as the DATALINK field, then use the DLVALUE function to change the character string to a DATALINK, as shown in Example 4-112.

Example 4-112 Updating a DATALINK using a parameter marker

```
// Create and open a connection to the iSeries.
iDB2Connection cn = new
iDB2Connection("DataSource=myiseries;DefaultCollection=sampledb;");
cn.Open();

// Create a command which updates the DATALINK field
// in the EMP_PHOTO table.
iDB2Command cmd = new iDB2Command("UPDATE EMP_PHOTO SET DL_PICTURE =
DLVALUE(CAST(@dl_picture AS CHAR(1000))) WHERE EMPNO='000010'", cn);

// Derive the parameter information.
cmd.DeriveParameters();

// Set the parameter value to a new datalink value.
cmd.Parameters[0].Value =
"HTTP://LP126AB.RCHLAND.IBM.COM/QIBM/ProdData/OS400/SQL/Samples/db200130.bmp";

// Execute the command.
cmd.ExecuteNonQuery();

// Dispose the command since we're done with it.
cmd.Dispose();

// Close the connection once more.
cn.Close();
```

Note: In this example, our Update statement casts the @dl_picture parameter as a CHAR(1000) because the DL_PICTURE field is a DATALINK with a length of 1000.

After completing these two latest examples, you may want to delete the row we just inserted and updated in the EMP_PHOTO table. Use the following command to delete the row:

```
DELETE FROM SAMPLEDB.EMP_PHOTO WHERE EMPNO='000010'
```

4.7.6 Connection pooling

Connection pooling is supported by many data providers. It enables similar connection objects to be pooled to reduce startup time, which is especially important with applications that open and close connections repeatedly, such as Web server applications. Because starting a connection to the server where your database resides (in our case, the iSeries server) can be a time-consuming process, the IBM.Data.DB2.iSeries .NET provider pools connections by default. Here is an example.

You have an application that resides on a Web server. Your application could get called by many different clients through their Web browsers. Further, your application reads data from a table on your iSeries and returns that data to the client. This scenario is shown in Figure 4-31.

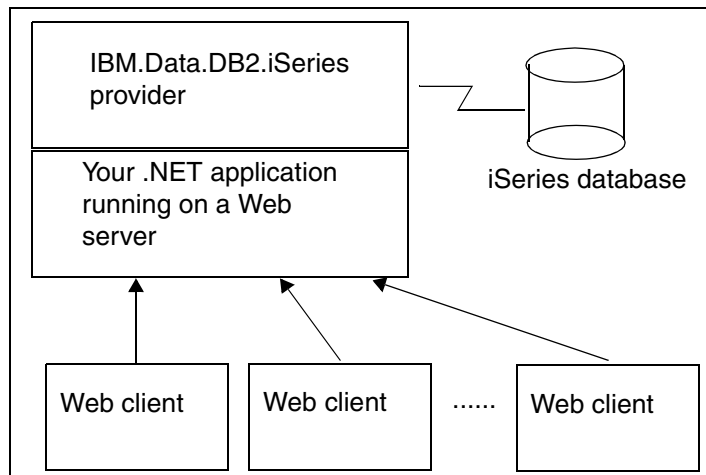


Figure 4-31 Web client/server scenario where connection pooling can help performance

If many clients want to use your data (from the iSeries), be sure that each client's connection to the iSeries is as short as possible. Your application should connect only to the iSeries long enough to send or receive the data it needs; that way, other clients will not get stuck waiting for an available connection. Because connection pooling makes starting and stopping a connection faster, there is no performance penalty each time you start or stop a connection.

Connection pooling is enabled by default (the iDB2Connection object's Pooling property turns pooling on or off; see "Connection pooling" on page 54 and the related ConnectionString properties that follow that section). The provider keeps track of pooled connections by keeping all connections with an identical ConnectionString in the same pool. In Example 4-113 on page 144, we create four connections. The connections whose ConnectionStrings are identical are in the same connection pool (c1a and c1b are in one pool, and c2a and c2b are in a different pool). Note that even though connectionString1 and connectionString2 are *logically* the same, they are not *identical* because connectionString2 has a blank space before the word DefaultCollection. Because the two strings are not identical, they cause the connections to go into a different connection pool.

Example 4-113 Example showing which connections will be in the same connection pool

```
// Our sample uses two different ConnectionStrings.
String connectionString1 = "DataSource=myiseries;DefaultCollection=sampled;";
String connectionString2 = "DataSource=myiseries; DefaultCollection=sampled;";

// Create two connections in the first connection pool
iDB2Connection c1a = new iDB2Connection(connectionString1);
iDB2Connection c1b = new iDB2Connection(connectionString1);

// Create two connections in the second connection pool
iDB2Connection c2a = new iDB2Connection(connectionString2);
iDB2Connection c2b = new iDB2Connection(connectionString2);

// Now, open the two connections in the first connection pool.
// Each connection has its own iSeries server job.
c1a.Open();
c1b.Open();
Console.WriteLine("Connection 1a's JobName: " + c1a.JobName);
Console.WriteLine("Connection 1b's JobName: " + c1b.JobName);

// Now, run a command that causes an error on connection 1a.
// We do this so we'll get an entry in our job log.
iDB2Command cmd = new iDB2Command("inserrrrrrrrrt into employee", c1a);
try
{
    cmd.ExecuteNonQuery();
}
catch
{
    // Ignore errors.
}
cmd.Dispose();

// Now, close connection 1a, and then reopen it.
c1a.Close();
c1a.Open();

// Display the job name. Because the Close() method
// placed the connection back into the connection pool,
// Open() could use that connection again.
// You will see the JobName is the same as the job
// it used previously.
Console.WriteLine("Connection 1a's JobName after close/open: " + c1a.JobName);

// Now, go and look at job 1a's job log on the iSeries.
// You should see the SQL0104 error in the job log, since
// the connection was reused.
Console.WriteLine("Go and look at this joblog on the iSeries host now: " + c1a.JobName);
Console.WriteLine("Press the enter key when you are finished.");
Console.ReadLine();

// Close the connections.
c1a.Close();
c1b.Close();

// Now, open the connections in the second connection pool.
// Because the ConnectionString used by c2a and c2b are
// different from the ConnectionString used by c1a and c2b,
// they will use different iSeries server jobs.
c2a.Open();
```

```
c2b.Open();
Console.WriteLine("Connection 2a's JobName: " + c2a.JobName);
Console.WriteLine("Connection 2b's JobName: " + c2b.JobName);

// Close the connections
c2a.Close();
c2b.Close();
```

When you close a pooled connection, it is released into the connection pool to be reused. Then, when a new connection wants to open a connection to the iSeries and uses the same `ConnectionString` as a connection in the pool, it gets a pooled connection and reduced startup time.

When using connection pooling, the first time you open a pooled connection there is no available connection in the pool, so the provider creates one for you. That means that the first time you call the `Open` method of a connection, the startup time takes longer, but each subsequent time that connection is closed and reopened, the startup time is reduced.

When pooled connections go bad

Sometimes, for whatever reason, your connection to the iSeries host is ended. This can have an effect on pooled connections. Read about how to handle this condition in “Special considerations when using connection pooling” on page 105.

When pooled connections stay around too long

Just as there are times when pooled server jobs terminate unexpectedly, sometimes server jobs stay running even after your application has terminated. This could happen for several reasons, such as when your application does not clean up all of its resources properly or because of host server or communication settings.

The .NET runtime environment uses garbage collection and finalizers to ensure that application resources are reclaimed eventually. However, garbage collection cannot handle some things, such as unmanaged resources and resources on the server. The runtime environment does not always give class finalizers a chance to run to completion, so you should be in the habit of calling the `Close` or `Dispose` method of objects you no longer need.

In some cases, you may find that even though you are doing all of the cleanup that you think is necessary, the host server jobs are still running. To remedy this situation, the `IBM.Data.DB2.iSeries` provider has implemented a method called `CleanupPooledConnections`. This is a static method in the `iDB2ProviderSettings` class.

Note: The `iDB2ProviderSettings.CleanupPooledConnections()` method was first added to the provider in V5R3M0 service pack SI15176.

Calling *CleanupPooledConnections*

`CleanupPooledConnections` should *only* be called when you are finished using all of your `iDB2Connections` (for example, when your application is about to end). It causes all of your application's pooled connections to terminate their connection to the iSeries host and marks them as invalid. You should not attempt to use any `iDB2Connection` in your application after calling `CleanupPooledConnections`.

Note: See 4.10.7, “Gathering information for IBM Support” on page 169 for information about why you might want to call `CleanupPooledConnections` when you are doing problem determination.

Example 4-114 shows how to call this method when your Windows application receives the Cancel event. (This happens when your application terminates, such as when you press the X icon to close it). It first declares an event handler for the Cancel event, then defines the event handler that calls `CleanupPooledConnections`.

Example 4-114 CleanupPooledConnections method

```
// Add this to the public Form1() method after it calls InitializeComponent():
this.Closing += new CancelEventHandler(this.Form1_Cancel);

...
// Add this new method near the end of your source file:
protected void Form1_Cancel(object sender, System.ComponentModel.CancelEventArgs e)
{
    iDB2ProviderSettings.CleanupPooledConnections();
}
```

You can add the code shown in Example 4-114 to any of your existing Windows applications (for example, 4.6.2, “A simple DataAdapter with CommandBuilder example” on page 110 or 4.7.3, “Updating DataSets” on page 136).

4.7.7 Deploying your application

When it is time to deploy your .NET application for use with the IBM.Data.DB2.iSeries .NET provider, you must consider the following items:

- ▶ You must install the .NET Framework onto your PC *before* you install the provider.
- ▶ The provider is not stand-alone. Because the IBM DB2 UDB for iSeries .NET provider is part of the iSeries Access for Windows product and relies on its features, you must install it using iSeries Access for Windows on each PC that you plan to use the .NET provider from. Read more about installing in 4.3, “Before we begin” on page 39. iSeries Access for Windows provides a robust, configurable set-up program that enables you to install only the necessary components.
- ▶ Because changes to the provider are sometimes tied in with changes to other parts of iSeries Access for Windows, try to avoid running a back-level provider with a newer version of iSeries Access for Windows. This back-leveling could occur, for example, if you first install iSeries Access for Windows plus a service pack, then later install the provider using Selective Setup. If you do this, be sure to (re)apply the iSeries Access for Windows service pack after installing the provider to ensure that all components have the most recent updates applied.
- ▶ Be sure to test your application with different iSeries versions. If you try to use a feature that is only available in a newer DB2 UDB for iSeries version (such as the Binary data type or 63-digit decimals added to the iSeries in V5R3M0), you will run into problems trying to use those features on an older iSeries version. See Example 4-34 on page 63 for an example of how you can programmatically check your iSeries server version.

4.8 Coding for performance and best practices

In this section, we discuss some of the things you can do to make the best use of the IBM DB2 UDB for iSeries provider.

- ▶ Read about iSeries database performance topics in the IBM Information Center at:

<http://www.iseries.ibm.com/infocenter>

Select **Database** → **Performance and optimization**.

- ▶ Read the “Indexing and statistics strategies for DB2 UDB for iSeries” white paper at:
<http://www.ibm.com/servers/enable/site/education/ibo/record.html?indxng>
- ▶ Attend the DB2 Performance Workshop; information at:
<http://www.ibm.com/servers/eserver/iseries/service/igs/db2performance.html>
- ▶ Study the DB2 UDB for iSeries online courses found at:
<http://www.ibm.com/servers/enable/site/education/ibo/view.html?oc#db2>
- ▶ Use stored procedures to group many operations into a single call. This can reduce the number of times data flows across the communication link. You can also use stored procedures to wrap programs or CL commands to catalog the parameter definitions so `DeriveParameters()` can be used on the call. For information about how to write stored procedures in the iSeries, go to the IBM Information Center at:
<http://www.iseries.ibm.com/infocenter>
Select **Database** → **Programming** → **SQL Programming** → **Routines** → **Stored Procedures**.
Another good reference is the IBM Redbook *Stored Procedures, Triggers and User Defined Functions on DB2 Universal Database for iSeries*, SG24-6503, at:
<http://www.redbooks.ibm.com/abstracts/sg246503.html>
- ▶ When using stored procedures, follow the guidelines spelled out in 4.5.5, “Calling stored procedures” on page 79. Using stored procedures with `CommandType.Text` and including your own parameter markers speeds up the time it takes to prepare the statement for execution.
- ▶ If you plan to execute a particular statement more than once, you can do several things to improve the time it takes to prepare and execute the statement:
 - a. Use parameter markers in your statement instead of literal values. Then, each time you execute the statement, your `CommandText` stays the same and only the parameter values change. This approach enables the iSeries server to optimize the statement once and subsequent calls to execute the statement take advantage of the optimization already done. Parameter markers are discussed in 4.5.4, “Using parameters in your SQL statements” on page 74. To take full advantage of this optimization, be sure your statement does not contain any literal values.
 - b. Keep the `CommandText` the same for a statement you plan to execute many times. While you can reuse the same `iDB2Command` object for running different SQL statements, do this only when you do not plan to re-execute a statement. Each time you change a command’s `CommandText`, the statement must be re-prepared before it is executed (the provider does this for you). By keeping the `CommandText` constant, you avoid this extra step.
- ▶ Use tables whose columns are not nullable. When using the `CommandBuilder`, every nullable column requires an extra check when performing updates and deletes. See 4.7.4, “Using `iDB2CommandBuilder`” on page 139 for more information about using the `CommandBuilder`. Also, when using a `DataReader` with columns that could contain null data, an extra call to `IsDBNull` is required to check for null before calling any of the `DataReader`’s `Get` methods.
- ▶ If your application does not have to use all columns of a table, do not select all of the columns. Fewer selected columns means less data sent over the communication link. The exception to this is when you use the `CommandBuilder`, because with `CommandBuilder`, your `Select` statement must include all of the primary or unique key columns.
- ▶ If your application does not have to use all the rows of a table, do not select all of the rows. Fewer rows selected means less data sent over the communication link.

- ▶ If you plan to make frequent calls to fill a `DataAdapter`, it may make more sense to open and close the connection yourself, rather than letting the `DataAdapter` do it for you. Although connection pooling helps improve connection startup performance, closing and reopening a connection causes the provider to re-prepare any commands it may have prepared on the previously open connection.
- ▶ Connection pooling, which is enabled by default, can help improve your performance. To take advantage of connection pooling, each connection you wish to pool must use exactly the same `ConnectionString`. See 4.7.6, “Connection pooling” on page 143.
- ▶ RLE compression, which is enabled via the `DataCompression` attribute in your `ConnectionString`, is turned on by default. RLE compression normally helps application performance, but some overhead is involved in compressing and decompressing data. By testing your application with your normal application data, you may find that setting `DataCompression` one way or the other can produce better performance.
- ▶ When using large objects (LOBs) or large data sets, consider using a `DataReader` instead of a `DataAdapter`. With a `DataReader`, you have more control over whether, when, and how you read your data. For more about using LOBs with the `IBM.Data.DB2.iSeries` provider, see 4.7.2, “Using large objects (LOBs)” on page 132.
- ▶ When using LOBs with a `DataReader` or with your own insert, update, or delete commands, you can optimize when the LOB data is read or written by using the `MaximumInlineLobSize` property. See “`MaximumInlineLobSize`” on page 58.
- ▶ Before executing your command, you do *not* have to call `Prepare()` first, as you may be used to doing with other database technologies. The provider always does an implicit prepare on your behalf when one of the `Execute` or `DeriveParameters` methods is called.
- ▶ Because the provider always does an implicit prepare when needed, you do *not* see a performance improvement by defining your own parameters. Instead, we encourage the use of the `iDB2Command` object’s `DeriveParameters()` method.
- ▶ The .NET common language runtime relies on garbage collection to handle the cleanup of objects. This garbage collection is non-deterministic; it happens at unspecified times. To make the best use of your application’s resources, you should call an object’s `Close()` or `Dispose()` method (when one exists) when you are finished using the object. This ensures that resources associated with your object are released when you know they are no longer needed. This is especially important with objects that connect to the host (`iDB2Connection`, `iDB2Command`, `iDB2DataReader`, and `iDB2DataAdapter`).
- ▶ Choose the best `Execute` method for your SQL statement. See 4.5.6, “Choosing your execute method” on page 86 for more information. Do not use `ExecuteReader` if `ExecuteNonQuery` or `ExecuteScalar` will do.
- ▶ With the `IBM.Data.DB2.iSeries` provider, normally you do not have to specify a `Database` property in your `ConnectionString` (see “`Database`” on page 51). If you are not using an independent auxiliary storage pool (IASP) with your application, then do not use the `Database` property.
- ▶ Use data types that are appropriate to your task. When using char data tagged with CCSID 65535, see “`iDB2CharBitData` and `iDB2VarCharBitData`” on page 90.
- ▶ When using a `CommandBuilder`, follow the guidelines discussed in 4.7.4, “Using `iDB2CommandBuilder`” on page 139.
- ▶ Under normal circumstances, do not enable tracing or diagnostics. Any time these are enabled, extra processing time and computer resources are used to gather tracing and diagnostic information. When you are finished with problem determination, disable your traces and diagnostics.
- ▶ Avoid using special characters in your SQL names (for instance, in table and column names). To include mixed upper-case and lower-case names in tables and columns, and

to include special characters in the names, some names must be delimited using quote characters. We call these types of identifiers *delimited identifiers*. Using delimited identifiers requires special processing on both the PC client and on the iSeries server. The extra processing is not large, but their use can be problematic. Whenever possible, you should also avoid using the three codepoints reserved as alphabetic characters for national languages (#, @, and \$ in the United States). These three codepoints should be avoided because they represent different characters, depending on the CCSID.

- ▶ Use try/catch blocks to catch exceptions that could occur in your application. This is especially important in places where object cleanup must occur in case of an error.
- ▶ Before calling any Get method of a DataReader (for example, GetDB2String) for a column that could contain a null value, call the IsDBNull method to check the column for a null value. If IsDBNull returns *true*, the column is null and you should not call any Get method, or an exception will result.

4.9 Migrating from ADO and OLE DB to ADO.NET

The ADO.NET architecture is different from other database provider architectures such as ADO, OLE DB, ODBC, and JDBC. Complete coverage of the differences between these database access technologies is beyond the scope of this book, but in this section we discuss some of these differences and show how some of the iSeries Access for Windows OLE DB provider functions previously accessed through Visual Basic and ADO can be performed using ADO.NET with the IBM.Data.DB2.iSeries .NET provider.

4.9.1 ADO objects and how they map to ADO.NET objects

In the ADO object model, you perform database operations through an OLE DB provider, such as IBMDBA400, IBMDBASQL, or MSDBASQL. The ADO objects are accessed via objects prefixed with *ADODB*. Table 4-8 shows how the most common ADO objects map to ADO.NET objects that you can use with the IBM.Data.DB2.iSeries .NET provider.

Table 4-8 Mapping ADO objects to ADO.NET objects included with the IBM.Data.DB2.iSeries provider

ADO object	Corresponding ADO.NET object
Command	iDB2 Command
<i>Command.Parameters collection</i>	<i>iDB2Command.Parameters (iDB2ParameterCollection)</i>
<i>Command.Properties collection</i>	<i>iDB2Command properties</i>
Connection	iDB2Connection
Connection.Errors collection	iDB2Exception.Errors (iDB2ErrorCollection)
Connection.Properties collection	iDB2Connection properties
Error	iDB2Error (item in an iDB2ErrorCollection)
Field	DataColumn of a DataRow, or column of a DataReader
Field.Properties collection	Column metadata returned from the DataReader's GetSchemaTable method or the DataAdapter's FillSchema method
Parameter	iDB2Parameter

ADO object	Corresponding ADO.NET object
Record	DataRow of a DataTable, or “current” row of a DataReader
Record.Fields collection	DataColumns of a DataRow, or columns of a DataReader
Recordset	iDB2DataReader for read-only, forward-only iDB2DataAdapter with DataTable for updatable
Recordset.Fields collection	DataColumns of a DataRow, or columns of a DataReader
Recordset.Properties collection	iDB2DataReader properties or iDB2DataAdapter properties

4.9.2 ADO recordsets versus ADO.NET DataReaders and DataAdapters

With ADO, database providers work using a connected paradigm in which you open a connection, work with recordsets, then close your connection. They give you a lot of control over how you read your data, and they have the concept of a *cursor* or current location within your result set. Your ADO recordset can use different types of cursors (such as forward-only, read-only, or updatable). When you work with your recordset, you can see the data in real time using a dynamic, sensitive cursor, or you can see a snapshot of your data using a static cursor. To read data using cursors, the database provider must stay connected to the iSeries server, so as you navigate forward or backward through your recordset, the provider reads the data from the server.

With ADO.NET, you can still navigate in a forward-only, read-only manner using a connected object called a DataReader. However, with a DataAdapter, your entire result set is read and placed into a DataSet. After the DataAdapter reads the data, it can disconnect and work with the DataSet locally. Changes to the data are made to the DataSet, and when you are ready to finalize your changes, you use the DataAdapter’s Update() method. Update causes the changes to your DataSet to be transmitted to the iSeries server. The DataAdapter only has to be connected to the iSeries server while it reads or updates data. The DataAdapter concept is somewhat similar to the ADO concept of a *client cursor*.

4.9.3 Updating tables

With the iSeries Access for Windows OLE DB providers IBMDBA400 and IBMDBASQL, you can update tables on an iSeries server using an updatable cursor. When updatable cursor is used, the OLE DB provider locks the current row. During changes to the result set (such as by calling recordset.Update or recordset.Delete from Visual Basic), the current row is locked to prevent concurrency violations. However, you might get an error if you try to navigate to a row that is locked by another application. This model is called *pessimistic concurrency*.

The IBM DB2 UDB for iSeries .NET provider uses optimistic concurrency to resolve changes to the iSeries database when you call the Update method of a DataAdapter. With optimistic concurrency, records are not locked. Instead, the provider generates statements that compare the original values read from the database with the current values in the database. If the original and current values differ, then a concurrency violation occurs, indicating that another job has modified the data in the table after you read the data. Read more about this topic in “Updating a DataSet with optimistic concurrency” on page 138.

4.9.4 Mapping OLE DB properties to ADO.NET

Migrating an application from using OLE DB to using the IBM.Data.DB2.iSeries .NET provider requires numerous changes. The .NET provider does not support Data Queues, Record Level Access, or certain other features that the iSeries Access for Windows OLE DB providers support, so you cannot use the .NET provider *in some cases*. However, in a large number of cases, you can migrate from OLE DB to .NET. Table 4-9 shows how some commonly used OLE DB connection properties are reflected in the DB2 UDB for iSeries .NET provider.

Table 4-9 Mapping OLE DB connection properties to IBM.Data.DB2.iSeries .NET provider properties

IBMDA400 or IBMDASQL OLE DB provider property	IBM.Data.DB2.iSeries .NET provider
Add Statements To SQL Package	Not applicable. The .NET provider does not support SQL packages.
Block Fetch	Not applicable. In the IBM.Data.DB2.iSeries provider, you cannot specify a block size for transferring data to and from the iSeries.
Catalog Library List	Not applicable. The IBM.Data.DB2.iSeries provider does not include catalog functions.
ConnectionTimeout	ConnectionTimeout property (see “ConnectionTimeout” on page 50).
Convert Date Time To Char	Not implemented by the IBM.Data.DB2.iSeries provider. Read about handling these values in “iDB2Date, iDB2Time, and iDB2TimeStamp” on page 94.
Current Catalog	Database property (see “Database” on page 51).
Cursor Sensitivity	Not applicable. The IBM.Data.DB2.iSeries provider does not expose cursor functions.
Data Compression	DataCompression property (see “DataCompression” on page 57).
Data Source	DataSource property (see “DataSource” on page 49).
DBMS Version	ServerVersion property (see “ServerVersion” on page 62).
Default Collection	DefaultCollection property (see “DefaultCollection” on page 52).
Force Translate	Not implemented by the IBM.Data.DB2.iSeries provider. Read about handling character data tagged with CCSID 65535 in “iDB2CharBitData and iDB2VarCharBitData” on page 90.
Hex Parser Option	HexParserOption property (see “HexParserOption” on page 58).
Initial Catalog	Database property (see “Database” on page 51).
Job Name	JobName property (see “JobName” on page 63).
Maximum Decimal Precision	MaximumDecimalPrecision property (see “MaximumDecimalPrecision” on page 60).
Maximum Decimal Scale	MaximumDecimalScale property (see “MaximumDecimalScale” on page 60).
Minimum Divide Scale	MinimumDivideScale property (see “MinimumDivideScale” on page 60).
Password	Password property (see “Password” on page 49).

IBMDA400 or IBMDASQL OLE DB provider property	IBM.Data.DB2.iSeries .NET provider
Provider	Provider property (see “Provider” on page 62). With the IBM.Data.DB2.iSeries .NET provider, you do not specify the Provider property in your ConnectionString.
Query Options File Library	QueryOptionsFileLibrary property (see “QueryOptionsFileLibrary” on page 59).
SSL	SSL property (see “SSL” on page 50).
Sort Language ID	SortLanguageId property (see “SortLanguageId” on page 57).
Sort Sequence	SortSequence property (see “SortSequence” on page 56).
Sort Table Name	SortTable property (see “SortTable” on page 57).
SQL Package Library Name	Not applicable. The .NET provider does not support SQL packages.
SQL Package Name	Not applicable. The .NET provider does not support SQL packages.
State	State property (see “State” on page 62).
Trace	Trace property (see “Trace” on page 63).
Unusable SQL Package Action	Not applicable. The .NET provider does not support SQL packages.
Use SQL Packages	Not applicable. The .NET provider does not support SQL packages.
User ID	UserId property (see “UserId” on page 49).

4.9.5 Examples showing an OLE DB application rewritten to use ADO.NET

In this section, we show two examples of existing applications that use the IBMDA400 OLE DB provider from iSeries Access for Windows. The applications are written in Visual Basic using ADO. We first show the original Visual Basic/ADO application, then we show the same application written in Visual Basic .NET using ADO.NET.

Forward-only, read-only example: ADO and ADO.NET

In this section, we select records from a table and read the records in forward-only, read-only.

Forward-only, read-only recordset example using ADO

In this example, we start with a Visual Basic application using ADO and the IBMDA400 OLE DB provider. The application reads from the EMPLOYEE table in the SAMPLEDB schema we set up in 1.4, “DB2 UDB for iSeries sample schema” on page 8. The application reads one record (row) at a time from the EMPLOYEE table and displays fields (columns) from that row in the window.

To re-create this sample ADO application:

1. Open a new Standard EXE project from Visual Basic. (We use Visual Basic 6.0.) Make sure the Project Explorer window is displayed (**View** → **Project Explorer**) and display the Toolbox (**View** → **Toolbox**).
2. Add a project reference (**Project** → **References**) to Microsoft ActiveX Data Objects 2.5 Library. Select the box next to the ADO library and click **OK**.

Note: You can use any ActiveX Data Object (ADO) library that is 2.5 or greater.

3. In the Project Explorer window of Visual Basic, right-click **Form1** and select **Properties**. Change the Caption to ForwardOnlyReadOnly.
4. In the Project Explorer window of Visual Basic, right-click **Project1** and select **Properties**. Change the Project Name to ForwardOnlyReadOnly.
5. In the Visual Basic design window, re-create the window design shown in Figure 4-32. The regular text fields are Labels, the white blank fields are Textboxes, and the four buttons on the bottom of the window are Buttons. Each of these items can be dragged from the Toolbox onto your window, and repositioned and resized as needed.

Figure 4-32 ForwardOnlyReadOnly window design (ADO)

6. Rename each of the Textboxes in your window to the names shown in parentheses in Figure 4-32. (For example, the first Textbox is called empno, the second is called empname, and so on.) Set the Text for each of these Textboxes to an empty string.
7. Rename each of the Buttons in your window to the names shown in parentheses in Figure 4-32. (For example, the leftmost button is called ConnectButton, the second is called DisconnectButton, and so on.) Change the Caption for each button to match our example. Set each button's Enabled property to **Disabled**, except for the Connect button. Leave the Connect button **Enabled**.
8. Add code for the application: Double-click anywhere on your application's window to display Form1's code window. Remove all code from the code window, and replace it with the code shown in Example 4-115, substituting the name of your iSeries.

Example 4-115 ForwardOnlyReadOnly code (ADO)

```

Private cn As New ADODB.Connection
Private cmd As New ADODB.Command
Private rs As New ADODB.Recordset
'Connect to the iSeries using the IBMDBA400 provider
'if we are not already connected
Private Sub ConnectButton_Click()
If (cn.State = adStateClosed) Then
    cn.ConnectionString = "Provider=IBMDA400; Data Source=myiseries; Default
Collection=sampledb;"
    cn.Open

    'Initialize the command to select records
    'from the EMPLOYEE table.
    'Point the command to our connection object

```

```

cmd.CommandText = "select * from employee"
cmd.ActiveConnection = cn

'Enable the disconnect and read buttons, and
'disable the connect button
DisconnectButton.Enabled = True
ReadButton.Enabled = True
ConnectButton.Enabled = False
End If
End Sub
'Disconnect from the iSeries if we are connected
Private Sub DisconnectButton_Click()
If (cn.State = adStateOpen) Then
    cn.Close

    'Disable all buttons except Connect.
    DisconnectButton.Enabled = False
    ReadButton.Enabled = False
    NextButton.Enabled = False
    ConnectButton.Enabled = True

    'Clear the fields on the window
    Form1.empno.Text = ""
    Form1.empname.Text = ""
    Form1.dept.Text = ""
    Form1.phone.Text = ""
    Form1.job.Text = ""
    Form1.edlevel.Text = ""
    Form1.salary.Text = ""
    Form1.bonus.Text = ""
    Form1.commission.Text = ""
    Form1.Show
End If
End Sub

'This routine moves to the next record in the table,
'and shows the new data. If we reach the end of the file,
'start reading again from the beginning.
Private Sub NextButton_Click()

'Make sure we have an open recordset first
If (rs.State = adStateOpen) Then
    'Move to the next record
    rs.MoveNext

    'If we are at EOF, restart at the beginning.
    If (rs.EOF = True) Then
        'We're at EOF. Restart at the beginning.
        Call ReadButton_Click
    Else
        'Display the results of the current row
        Call fillData
    End If
End If
End Sub

Private Sub ReadButton_Click()

'If we already have a recordset open,
'close it and then reopen it.

```

```

'Forward-only recordsets cannot be re-queried.
If (rs.State = adStateOpen) Then
    rs.Close
End If

'Create a recordset object which contains the results
'of executing our SELECT statement
Set rs = cmd.Execute

'Fill the window with the data from the first record
Call fillData

'Enable the Next button
NextButton.Enabled = True

End Sub
'This routine fills the text fields on our form
'using the data from our recordset
Private Sub fillData()
    Form1.empno.Text = rs.Fields("EMPNO")
    Form1.empname.Text = rs.Fields("LASTNAME") & ", " & _
        rs.Fields("FIRSTNAME") & " " & _
        rs.Fields("MIDINIT")
    Form1.dept.Text = rs.Fields("WORKDEPT")
    Form1.phone.Text = rs.Fields("PHONENO")
    Form1.job.Text = rs.Fields("JOB")
    Form1.edlevel.Text = rs.Fields("EDLEVEL")
    Form1.salary.Text = rs.Fields("SALARY")
    Form1.bonus.Text = rs.Fields("BONUS")
    Form1.commission.Text = rs.Fields("COMM")
    Form1.Show
End Sub

```

9. Using **File** → **Save Form1 As**, save your form as ForwardOnlyReadOnly.frm.
10. Using **File** → **Save Project As**, save your project as ForwardOnlyReadOnly.vbp.
11. Run your application (press F5 or **Run** → **Start**). Click **Connect** to connect to your iSeries. You may be prompted for your user ID and password. To open the recordset or to start reading again from the first record, click **Read data**. The first record from the EMPLOYEE table is displayed. To display each next record, click **Move to next row**. When you reach the end of the table, it wraps around and reads from the beginning again. When you are finished, click **Disconnect**.

Forward-only, read-only recordset example using ADO.NET

Now we migrate the application to ADO.NET. Follow these steps to see how we change the application from Visual Basic to Visual Basic .NET, and from ADO to ADO.NET:

1. Open a new project (from Visual Studio .NET, select **File** → **New** → **Project**). Select **Visual Basic Projects, Windows Application**. Name the project ForwardOnlyReadOnly. Click **OK**. Make sure the Solution Explorer window is displayed (**View** → **Solution Explorer**), display the Properties window (**View** → **Properties Window**), and display the Toolbox (**View** → **Toolbox**).
2. Add a project reference to the IBM DB2 UDB for iSeries .NET provider (**Project** → **Add Reference**). Select **IBM DB2 UDB for iSeries .NET provider** and click **OK**.
3. In the Form1.vb design window of Visual Basic .NET, click **Form1** to make it active. In the Properties window, change the Text field to ForwardOnlyReadOnly.

4. In the Visual Basic .NET design window, re-create the window design shown in Figure 4-33. The regular text fields are Labels, the white blank fields are Textboxes, and the four buttons on the bottom of the window are Buttons. Each of these items can be dragged from the Toolbox onto your window, and repositioned and resized as needed.

Figure 4-33 *ForwardOnlyReadOnly window design (ADO.NET)*

5. Rename each of the Textboxes in your window to the names shown in parentheses in Figure 4-33. (For example, the first Textbox is called empno, the second is called empname, and so on.) Set the Text for each of these Textboxes to an empty string.
6. Rename each of the Buttons in your window to the names shown in parentheses in Figure 4-33. (For example, the leftmost button is called ConnectButton, the second is called DisconnectButton, and so on.) Change the Caption for each button to match our example. Set each button's Enabled property to **Disabled**, except for the Connect button. Leave the Connect button **Enabled**.
7. Add code for the application by double-clicking anywhere in your application's window to display Form1's code window with the Form1_Load subroutine displayed. Add the following statement to the top of your code window:

```
Imports IBM.Data.DB2.iSeries
```

Next, place your cursor after the end of the Form1_Load subroutine, and add the code shown in Example 4-116.

Example 4-116 *ForwardOnlyReadOnly code (ADO.NET)*

```
'Declare our global variables
Dim cn As New iDB2Connection()
Dim cmd As New iDB2Command()
Dim rs As iDB2DataReader

'Connect to the iSeries using the IBM.Data.DB2.iSeries provider
'if we are not already connected
Private Sub ConnectButton_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles ConnectButton.Click
If (cn.State = ConnectionState.Closed) Then
cn.ConnectionString = "DataSource=myiseries; DefaultCollection=sampledb;"
cn.Open()

'Initialize the command to select records
```

```

        'from the EMPLOYEE table.
        'Point the command to our connection object
        cmd.CommandText = "select * from employee"
        cmd.Connection = cn

        'Enable the disconnect and read buttons, and
        'disable the connect button
        DisconnectButton.Enabled = True
        ReadButton.Enabled = True
        ConnectButton.Enabled = False
    End If
End Sub

'Disconnect from the iSeries if we are connected
Private Sub DisconnectButton_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles DisconnectButton.Click
    If (cn.State = ConnectionState.Open) Then

        'Close the DataReader if it is open
        If (Not (rs Is Nothing)) Then
            rs.Close()
            rs = Nothing
        End If

        'Close the connection.
        cn.Close()

        'Disable all buttons except Connect.
        DisconnectButton.Enabled = False
        ReadButton.Enabled = False
        NextButton.Enabled = False
        ConnectButton.Enabled = True

        'Clear the fields on the window
        empno.Text = ""
        empname.Text = ""
        dept.Text = ""
        phone.Text = ""
        job.Text = ""
        edlevel.Text = ""
        salary.Text = ""
        bonus.Text = ""
        commission.Text = ""
    End If
End Sub

'This routine moves to the next record in the table,
'and shows the new data. If we reach the end of the file,
'start reading again from the beginning.
Private Sub NextButton_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles NextButton.Click
    'Move to the next record.
    'If we are at EOF, restart at the beginning.
    If (rs.Read() = False) Then
        'We're at EOF. Restart at the beginning.
        Call ReadButton_Click(System.DBNull.Value, New System.EventArgs())
    Else
        'Display the results of the current row
        Call fillData()
    End If
End Sub

```

```

Private Sub ReadButton_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles ReadButton.Click
    'If we already have a DataReader open,
    'close it and then reopen it.
    'DataReaders can only be read forward-only.
    If (Not (rs Is Nothing)) Then
        rs.Close()
    End If

    'Create a DataReader object which contains the results
    'of executing our SELECT statement
    rs = cmd.ExecuteReader()

    'Fill the window with the data from the first record
    If (rs.Read() = True) Then
        Call fillData()
    End If

    'Enable the Next button
    NextButton.Enabled = True
End Sub

'This routine fills the text fields on our form
'using the data from our recordset
Private Sub fillData()
    empno.Text = rs("EMPNO")
    empname.Text = rs("LASTNAME") & ", " & _
        rs("FIRSTNAME") & " " & _
        rs("MIDINIT")
    dept.Text = rs("WORKDEPT")
    phone.Text = rs("PHONENO")
    job.Text = rs("JOB")
    edlevel.Text = rs("EDLEVEL")
    salary.Text = rs("SALARY")
    bonus.Text = rs("BONUS")
    commission.Text = rs("COMM")
End Sub

```

8. Using **File** → **Save Form1.vb As**, save your form as ForwardOnlyReadOnly.vb.
9. Using **File** → **Save All**, save your solution.
10. Run your application (press F5 or **Run** → **Start**). Click **Connect** to connect to your iSeries. You may be prompted for your user ID and password. To open the recordset or to start reading again from the first record, click **Read data**. This displays the first record from the EMPLOYEE table. To display each next record, click **Move to next row**. When you reach the end of the table, it wraps around and reads from the beginning again. When you are finished, click **Disconnect**.

Comparison between our ADO example and our ADO.NET example

Our examples have shown few differences between the ADO and the ADO.NET code. Now we compare the ADO example to the ADO.NET example, showing how to convert the ADO example to ADO.NET.

1. Different object names:
 - With ADO, use ADODB objects (Connection, Command, Recordset).
 - With ADO.NET, use .NET provider objects (iDB2Connection, iDB2Command, iDB2DataReader).

2. The `ConnectionString` is different:
 - With ADO, you must specify the Provider name (Provider=IBMDA400).
 - With ADO.NET, the provider is implicit in the class you use. (iDB2Connection is the IBM.Data.DB2.iSeries provider.)
3. Command properties and methods are different:
 - With ADO, you use `command.ActiveConnection` and `command.Execute`.
 - With ADO.NET you use `command.Connection` and `command.ExecuteReader`.
4. Result data properties and methods are different:
 - With ADO, you use `recordset.EOF` and `recordset.MoveNext`.
 - With ADO.NET, you use `datareader.Read()`.
5. Enumeration values are different, such as when checking the Connection state:
 - With ADO, you use `adStateOpen`.
 - With ADO.NET you use `ConnectionState.Open`.
6. Differences in how you position to the first record:
 - With ADO, when you open a recordset, you are automatically positioned at the first record.
 - With ADO.NET, you must call the `DataReader's Read()` method before you can access the data from the first record.

Dynamic, updatable recordset example: ADO and ADO.NET

In this section, we show a scenario that uses an updatable recordset (ADO) and a `DataAdapter` (ADO.NET). We select records from a table, then read and write the records.

Updatable recordset example using ADO

We start with a Visual Basic application using ADO and the IBMDA400 OLE DB provider. The application reads from the ACT table in the SAMPLEDB schema we set up in 1.4, "DB2 UDB for iSeries sample schema" on page 8. The ACT table is small and easy to use, and it has a primary key (important for the next section when we update using a `CommandBuilder`). The application reads a record (row) from the ACT table and displays fields (columns) from that row on the window. You can then update the record that is displayed, delete the record, or add a new record.

To re-create this sample ADO application:

1. Open a new Standard EXE project from Visual Basic. (We use Visual Basic 6.0.) Make sure the Project Explorer window is displayed (**View** → **Project Explorer**) and display the Toolbox (**View** → **Toolbox**).
2. Add a project reference (**Project** → **References**) to Microsoft ActiveX Data Objects 2.5 Library. Select the box next to the ADO library and click **OK**.

Note: You can use any ActiveX Data Object (ADO) library that is 2.5 or greater.

3. In the Project Explorer window of Visual Basic, right-click **Form1** and select **Properties**. Change the Caption to `UpdatableSample`.
4. In the Project Explorer window of Visual Basic, right-click **Project1** and select **Properties**. Change the Project Name to `UpdatableSample`.
5. In the Visual Basic design window, re-create the window design shown in Figure 4-34 on page 160. The regular text fields are Labels, the white blank fields are Textboxes, and the

buttons are Buttons. Each of these items can be dragged from the Toolbox onto your window, and repositioned and resized as needed.

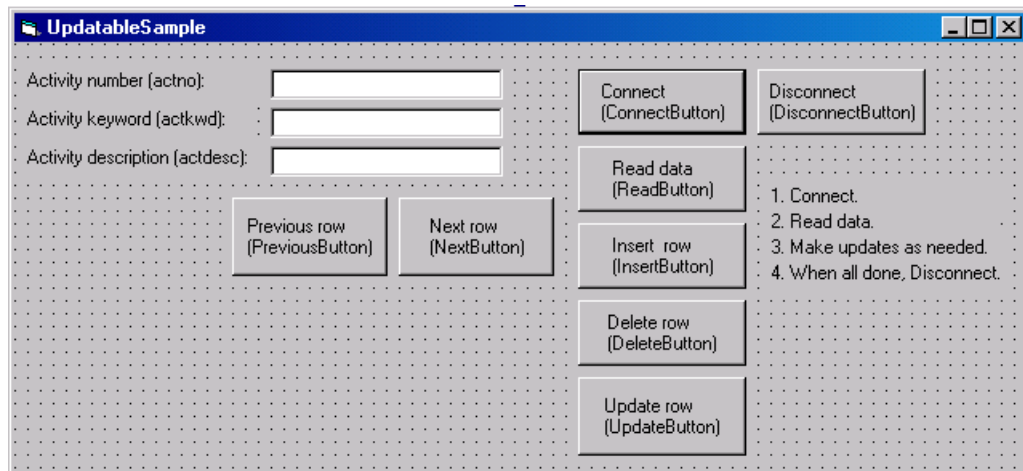


Figure 4-34 UdatableSample window design (ADO)

6. Rename each of the Textboxes in your window to the names shown in parentheses in Figure 4-34. (For example, the first Textbox is called `actno`, the second is called `actkwd`, and the third is called `actdesc`. Set the Text for each of these Textboxes to an empty string.
7. Rename each of the Buttons in your window to the names shown in parentheses on the buttons in Figure 4-34. Change the Caption for each button to match our example. Set each button's Enabled property to **Disabled**, except for the Connect button. Leave the Connect button **Enabled**.
8. Add code for the application by double-clicking anywhere in your application's window to display Form1's code window. Remove all code from the code window, and replace it with the code shown in Example 4-117.

Example 4-117 Udatable code (ADO)

```
Private cn As New ADODB.Connection
Private cmd As New ADODB.Command
Private rs As New ADODB.Recordset

'Connect to the iSeries using the IBMDA400 provider
'if we are not already connected
Private Sub ConnectButton_Click()
If (cn.State = adStateClosed) Then
    cn.ConnectionString = "Provider=IBMDA400; Data Source=myiseries; Default
Collection=sampled; "
    cn.Open

    'Initialize the command to select records
    'from the ACT table.
    'Point the command to our connection object
    cmd.CommandText = "select * from act"
    cmd.ActiveConnection = cn

    'Initialize the recordset properties so we
    'get an updatable cursor.
    rs.CursorType = adOpenDynamic
    rs.LockType = adLockPessimistic

    'Enable the Disconnect and Read buttons, and
```

```

        'disable the Connect button.
        ConnectButton.Enabled = False
        DisconnectButton.Enabled = True
        ReadButton.Enabled = True
    End If
End Sub

'Disconnect from the iSeries if we are connected
Private Sub DisconnectButton_Click()
    If (cn.State = adStateOpen) Then
        cn.Close

        'Disable all buttons except Connect.
        ConnectButton.Enabled = True
        DisconnectButton.Enabled = False
        ReadButton.Enabled = False
        PreviousButton.Enabled = False
        NextButton.Enabled = False
        InsertButton.Enabled = False
        UpdateButton.Enabled = False
        DeleteButton.Enabled = False

        'Clear the fields on the window
        showBlankForm
    End If
End Sub

Private Sub NextButton_Click()
    'If we are at EOF, move back to the beginning
    If (rs.EOF = True) Then
        rs.MoveFirst
    Else
        rs.MoveNext
        If (rs.EOF = True) Then
            rs.MoveFirst
        End If
    End If
    Call fillData
End Sub

Private Sub PreviousButton_Click()
    'If we are at BOF, move to the end
    If (rs.BOF = True) Then
        rs.MoveLast
    Else
        rs.MovePrevious
        If (rs.BOF = True) Then
            rs.MoveLast
        End If
    End If
    Call fillData
End Sub

Private Sub ReadButton_Click()

    'If we already have a recordset open,
    'move back to the top. Because this is a
    'dynamic, updatable cursor, we can do this.
    If (rs.State = adStateOpen) Then

```

```

        rs.MoveFirst
    Else
        'If we don't already have an open recordset,
        'open it now.
        rs.Open cmd
    End If

    'Fill the window with the data from the first record
    Call fillData

    'Enable the other buttons
    PreviousButton.Enabled = True
    NextButton.Enabled = True
    InsertButton.Enabled = True
    UpdateButton.Enabled = True
    DeleteButton.Enabled = True

End Sub

'Add the new row using the current form values and then
'position back to the top of the table
Private Sub InsertButton_Click()
    'We will update all the fields
    flds = Array("ACTNO", "ACTKWD", "ACTDESC")

    'Take the values from the form
    vals = Array(Form1.actno.Text, Form1.actkwd.Text, Form1.actdesc.Text)

    'Add the new row.
    rs.AddNew flds, vals
    rs.MoveFirst
    fillData
End Sub

'Update the current record and then
'position back to the top of the table
Private Sub UpdateButton_Click()
    Dim flds, vals As Variant

    'We will update all the fields
    flds = Array("ACTNO", "ACTKWD", "ACTDESC")

    'Take the values from the form
    vals = Array(Form1.actno.Text, Form1.actkwd.Text, Form1.actdesc.Text)

    'Update the row
    rs.Update flds, vals
    rs.MoveFirst
    fillData

End Sub

'Delete the current record and then
'position back to the top of the table
Private Sub DeleteButton_Click()
    'Delete the row
    rs.Delete
    rs.MoveFirst
    fillData
End Sub

'This routine fills the text fields on our form

```

```
'using the data from our recordset
Private Sub fillData()
    Form1.actno.Text = rs.Fields("ACTNO")
    Form1.actkwd.Text = rs.Fields("ACTKWD")
    Form1.actdesc.Text = rs.Fields("ACTDESC")
    Form1.Show
End Sub

Private Sub showBlankForm()
'Clear the fields on the window
    Form1.actno.Text = ""
    Form1.actkwd.Text = ""
    Form1.actdesc.Text = ""
    Form1.Show
End Sub
```

9. Using **File** → **Save Form1 As**, save your form as UpdatableSample.frm.
10. Using **File** → **Save Project As**, save your project as UpdatableSample.vbp.
11. Run your application (press F5 or **Run** → **Start**). Click **Connect** to connect to your iSeries. You may be prompted for your user ID and password. To open the recordset or to start reading again from the first record, click **Read data**. The first record from the ACT table is displayed. To display the next or previous record, click **Previous row** or **Next row**.

Because this is an updatable example, you can insert a new row, delete a row, or update an existing row:

- To insert a new row, position on any record. Change the fields in the window to correspond to the new row, and click on **Insert row**. The new row is sent to the iSeries host, and you are returned to the first row in the table. You must provide a unique value for the Activity number field because that is a primary key field.
- To delete an existing row, navigate to the row and click **Delete row**. The delete information is sent to the iSeries host, and you are returned to the first row in the table.
- To update an existing row, navigate to the row, make your changes to the form, and click **Update row**. Changes are sent to the iSeries host, and you are returned to the first row in the table.

When you are finished with your example, click **Disconnect**.

Updatable recordset example using ADO.NET

Now we migrate the application to ADO.NET. Follow these steps to see how we change the application from Visual Basic to Visual Basic .NET and from ADO to ADO.NET:

1. Open a new project (from Visual Studio .NET, select **File** → **New** → **Project**). Select **Visual Basic Projects, Windows Application**. For the project name, type UpdatableSample. Click **OK**. Make sure the Solution Explorer window is displayed (**View** → **Solution Explorer**), display the Properties window (**View** → **Properties Window**), and display the Toolbox (**View** → **Toolbox**).
2. Add a project reference to the IBM DB2 UDB for iSeries .NET provider (**Project** → **Add Reference**). Select **IBM DB2 UDB for iSeries .NET provider** and click **OK**.
3. In the Form1.vb design window of Visual Basic .NET, click **Form1** to make it active. In the Properties window, change the Text field to UpdatableSample.
4. In the Visual Basic .NET design window, re-create the window design shown in Figure 4-35 on page 164. The large area in the middle of the screen is a DataGrid and the buttons on the window are Buttons. Each of these items can be dragged from the Toolbox onto your window, and repositioned and resized as needed.

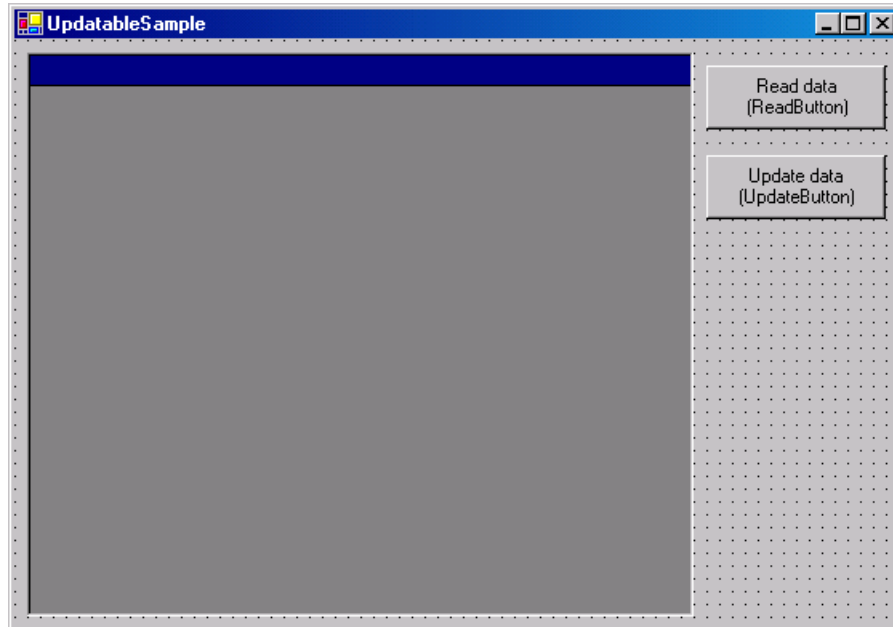


Figure 4-35 UdatableSample window design (ADO.NET)

5. Because our ADO.NET sample uses a DataGrid instead of Textboxes to display the data, we do not have to rename the Textbox fields as we did in our previous example.
6. Rename each of the Buttons in your window to the names shown in parentheses in Figure 4-35. Change the Caption for each button to match our example. Set the UpdateButton's Enabled property to **Disabled**. Leave the ReadButton **Enabled**.
7. Add code for the application by double-clicking anywhere in your application's window. This displays Form1's code window showing the Form1_Load subroutine. Add this statement to the top of your code window:

```
Imports IBM.Data.DB2.iSeries
```

Replace your Form1_Load subroutine with the code shown in Example 4-118.

Example 4-118 Udatable code (ADO.NET)

```
'Declare our global variables
Dim cn As New iDB2Connection()
Dim ds As New DataSet()
Dim da As New iDB2DataAdapter()
Dim cb As New iDB2CommandBuilder(da)

Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles MyBase.Load
    'Initialize our Connection string
    cn.ConnectionString = "DataSource=myiseries; DefaultCollection=sampled; "

    'Initialize the SelectCommand of the DataAdapter
    da.SelectCommand = New iDB2Command("select * from act", cn)

    'Tell the DataGrid where its data will come from
    DataGrid1.DataSource = ds
End Sub

Private Sub ReadButton_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles ReadButton.Click
```

```

        ds.Clear()
        da.Fill(ds)
        DataGrid1.DataMember = ds.Tables(0).TableName
        UpdateButton.Enabled = True
    End Sub

    Private Sub UpdateButton_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles UpdateButton.Click
        cn.Open()
        da.Update(ds)
        cn.Close()
    End Sub

```

8. Using **File** → **Save Form1.vb As**, save your form as UpdatableSample.vb.
9. Using **File** → **Save All**, save your solution.
10. Run your application (press F5 or **Run** → **Start**). Because this example uses a DataAdapter, you do not have to connect to read the data; the DataAdapter opens the connection for you. To fill the DataGrid with the data from our table using the DataAdapter, click **Read data**. The DataGrid is filled with data from the ACT table.

Because this is an updatable example, you can insert a new row, delete a row, or update an existing row. We use the DataAdapter to make our updates, so updates are stored locally in our DataSet until you click **Update data**.

- To insert a new row, position yourself after the last row and add data on the window to correspond to the new row. You must provide a unique value for the Activity number field because that is a primary key field.
- To delete an existing row, navigate to the row, highlight it, and press the Delete key.
- To update an existing row, navigate to the row, and make your changes to the form.

When you are finished making changes (inserts, updates, and deletions), click **Update data**. At this point, the connection to the iSeries is opened, the DataAdapter sends the updates to the iSeries host, and the connection is closed.

Comparison between our ADO example and our ADO.NET example

Unlike our forward-only read-only examples, our updatable examples show many differences and only a few similarities. In this section we compare the ADO example to the ADO.NET example and show you step-by-step how we convert the ADO example to ADO.NET.

1. Connected versus Disconnected mode:
 - With ADO, you must stay connected the whole time you want to access your data. Because of this, we have Connect and Disconnect buttons and event handlers.
 - With ADO.NET, you stay connected to the iSeries only while you are actively getting the data from the iSeries (DataAdapter.Fill) or sending updates to the iSeries (DataAdapter.Update).
2. Differences in how you read and store the result data:
 - With ADO, you use an updatable recordset to read the data, and you store the data in Textbox fields.
 - With ADO.NET, you use a DataAdapter to read the data into a DataSet, which is then associated with a DataGrid.
3. Differences in how you navigate through the result data:
 - With ADO, you must handle navigation yourself by using the ADODB recordset's methods such as MovePrevious, MoveNext, and MoveFirst. Because of this, we have Previous and Next buttons and event handlers.

- With ADO.NET, after you get the data into the DataGrid using the DataSet, all the navigation is handled for you.
4. Differences in how you update the data:
 - With ADO, the data is stored on the iSeries server. Each time you want to insert, update, or delete a row, the change is transmitted to the iSeries server immediately (when you press the corresponding button).
 - With ADO.NET, the DataSet/DataGrid contains a local cache of your data. All inserts, updates, and deletes are performed using this local cache, and changes are transmitted to the iSeries host only when you click Update.
 5. Different object names:
 - With ADO, you use the ADODB.Connection, Command, and Recordset objects.
 - With ADO.NET, you use the IBM.Data.DB2.iSeries provider's objects (iDB2Connection, iDB2DataAdapter, and iDB2CommandBuilder).
 6. The ConnectionString is different:
 - With ADO, you must specify the Provider property (Provider=IBMDA400).
 - With ADO.NET, the provider is implicit in the class you use. (iDB2Connection is the IBM.Data.DB2.iSeries provider.)
 7. Overall logic change: Because the DataAdapter, CommandBuilder, DataSet, and DataGrid do much of the work for you, using updatable recordsets with ADO.NET can be much easier than using ADO.

4.10 Troubleshooting

When you encounter problems in writing or running your .NET code using the IBM.Data.DB2.iSeries provider, you have several tools at your disposal. In this section, we offer some pointers to help you determine the cause of a failure, and information you should gather when you prepare to call your IBM Service representative.

4.10.1 Handle exceptions using try/catch blocks

The most common error surfaces as an exception of some kind. If the statement causing the exception is not surrounded by a try/catch block, an error pops up onto your display window with limited information about the error. For many exceptions, the error that pops up does not provide enough information to determine its cause. For more information about the exception, add a try/catch block around the exception. See 4.5.8, "Handling exceptions" on page 102 for more details about handling exceptions.

When catching exceptions, always catch the more specific exception first, followed by more generic exceptions. For example, say you want to execute this statement:

```
select * from employee
```

This statement could return an iDB2SQLException (for instance, if the table does not exist), a more generic iDB2Exception (for some other kind of error thrown by the provider), or a still more generic Exception. Example 4-119 on page 167 shows how you can catch these exceptions, in order from most-specific to least-specific error.

Example 4-119 Catching exceptions, from most-specific to least-specific

```
// Create and open a connection to the iSeries.
iDB2Connection cn = new
iDB2Connection("DataSource=myiseries;DefaultCollection=sampledb;");
cn.Open();

// Create a command to select rows from the table.
iDB2Command cmd = new iDB2Command("select * from employee", cn);
iDB2DataReader dr = null;

// Surround our Execute command with a try/catch block.
// We catch the most specific exceptions first, followed
// by the least specific exceptions.
try
{
    dr = cmd.ExecuteReader();
}
catch (iDB2SQLException e)
{
    Console.WriteLine("Message: " + e.Message);
    Console.WriteLine("MessageCode: " + e.MessageCode);
    Console.WriteLine("MessageDetails: " + e.MessageDetails);
    Console.WriteLine("SqlState: " + e.SqlState);
}
catch (iDB2Exception e)
{
    Console.WriteLine("Message: " + e.Message);
    Console.WriteLine("MessageCode: " + e.MessageCode);
    Console.WriteLine("MessageDetails: " + e.MessageDetails);
}
catch (Exception e)
{
    Console.WriteLine("Message: " + e.Message);
}

// Close the DataReader if it exists.
if (dr != null)
    dr.Close();

// Dispose the command since we are done using it.
cmd.Dispose();

// Close the connection.
cn.Close();
```

4.10.2 Make sure your server jobs are running

For connection-related problems, be sure your iSeries server jobs are started (see 4.2.2, “Host server jobs” on page 35). Read “Special considerations when using connection pooling” on page 105.

4.10.3 Use provider traces via the cwbmptrc utility

A useful tool in problem determination is the **cwbmptrc** utility, part of the iSeries Access for Windows product. Type **cwbmptrc** with no parameters to see a list of valid parameters. To turn traces on, type **cwbmptrc +a**. To turn traces off, type **cwbmptrc -a**. When you turn traces on or off, the **cwbmptrc** tool tells you where the trace file is located.

Important: After you no longer need to run traces, be sure to turn traces off using `cwbmptrc -a`. Otherwise, your trace file will grow larger each time you use any of the IBM.Data.DB2.iSeries provider classes. Tracing also slows down your performance.

After running traces while reproducing the problem, examine the `cwbmptrc` output file (the default file name is `idb2trace.txt`). You can often learn things from looking at a trace, including where the provider generates an exception.

The `cwbmptrc` command is located in the folder where you installed iSeries Access for Windows. If your PC's path does not point to this folder, you may need to navigate into that folder in order for the `cwbmptrc` command to run.

4.10.4 Enable server-side diagnostics

For more difficult problems, you can enable server-side diagnostics through use of the *Trace* property in your `ConnectionString`. See "Trace" on page 63 for more information.

4.10.5 Use communication traces via the `cwbcotrc` utility

Communication traces can be enabled on your PC using the `cwbcotrc` utility, part of the iSeries Access for Windows product. Type `cwbcotrc` with no parameters to see a list of valid parameters. To turn traces on, type `cwbcotrc on`. To turn traces off, type `cwbcotrc off`. When you turn traces on or off, the `cwbcotrc` tool tells you where the trace file is located.

Important: After you no longer need to run traces, be sure to turn traces off using `cwbcotrc off`. Otherwise, your trace file will grow larger each time you use any of the communication functions of the iSeries Access for Windows product. Tracing also slows down your performance. If your application is running in a three-tier environment, you may need to use the `/ALLUSERS` option to enable tracing for all applications.

The `cwbcotrc` command is located in the folder where you installed iSeries Access for Windows. If your PC's path does not point to this folder, you may have to navigate into that folder in order for the `cwbcotrc` command to run.

4.10.6 Overriding your `ConnectionString`

For problem determination, it is sometimes useful to change your `ConnectionString`, such as to enable server-side traces to spool the job log, or to connect to a different iSeries server during testing. In some cases, however, you cannot modify the `ConnectionString` (for example, if you only have an executable but no source code).

IBM has implemented a way for you to *override* your `ConnectionString`. When you use the `-override` feature of the `cwbmptrc` utility, the provider stores your *override string* into the system registry. To set the override, you must have write authority to the registry location `HKEY_LOCAL_MACHINE\SOFTWARE\IBM\Client Access\CurrentVersion\Common\ManagedProvider\Service\Connection String Override`.

Important: Because the override string is stored in the system registry, the override persists across reboots, and it affects all applications on your PC. Using the `ConnectionString` override should *only* be used as long as it is required; you should clear the override as soon as it is practical.

The `ConnectionString` override has the same format as your `ConnectionString` (see 4.5.2, “iDB2Connection and `ConnectionString` properties” on page 48). When an override is in place, any time an application on your PC opens a connection (using an `iDB2Connection` object’s `Open` method), the override string is appended to the `ConnectionString` of the connection object. The provider then parses the `ConnectionString` left-to-right, so if a duplicate keyword exists in the string, the rightmost one is used. Because the override string is *appended* to the `ConnectionString`, any keyword listed in both the `ConnectionString` and in the override string will use the value from the override string.

The `ConnectionString` override feature uses the following syntax:

- To set an override:

```
cwbmptrc -override "connection string override"
```

- To remove all overrides and return to the default processing:

```
cwbmptrc -override ""
```

- To show the current trace settings:

```
cwbmptrc -v
```

If an override has been set, **cwbmptrc -v** shows the override setting.

The **cwbmptrc** command is located in the folder where you installed iSeries Access for Windows. If your PC’s path does not point to this folder, you may have to navigate into that folder in order for the **cwbmptrc** command to run.

Table 4-10 shows some examples of `ConnectionString` overrides.

Table 4-10 Sample `ConnectionString` overrides

ConnectionString override	Explanation
cwbmptrc -override "DataSource=myiSeries; ConnectionTimeout=60;"	Overrides the <code>DataSource</code> used for the connection. Sets the <code>ConnectionTimeout</code> value.
cwbmptrc -override "Trace=PrintJoblog;Pooling=false;"	Enables the server job log to be spooled when the job ends. Disables connection pooling.
cwbmptrc -override "DataSource=as1; DataSource=myiSeries;"	Overrides the <code>DataSource</code> used for the connection. Because it is listed last, the <code>DataSource</code> used is <code>myiSeries</code> .
cwbmptrc -override ""	All overrides are canceled.

Note: The `ConnectionString` override feature was added to the provider in service pack SI13587. Because it was added after the product shipped, there is limited, English-only help text available for the **cwbmptrc -override** option in V5R3M0.

4.10.7 Gathering information for IBM Support

If you have exhausted your problem determination and believe that you have found a problem with the provider, it may be time to contact your IBM Support representative. When contacting IBM Support, include:

- The exact error message or output from the failing application. For `iDB2Exceptions`, include the `Message`, `MessageCode`, `SqlState`, and `StackTrace` portions.

Note: The `SqlState` property is meaningful only for `iDB2SQLExceptions`.

- ▶ The iSeries Access for Windows version and service pack level. Find the version and service level listed on the General tab of the iSeries Access for Windows Properties dialog by selecting **Start** → **Settings** → **Control Panel** → **iSeries Access for Windows**.
- ▶ The iSeries host version and cumulative PTF level. To find the host version and cumulative PTF level from an iSeries command prompt, run the DSPPTF command. The top PTF listed in Display PTF Status is a marker PTF that has a PTF ID listed in the TL0nnnn format. (nnnn is a four-digit number that is encoded with your current come tape level.)
- ▶ The iSeries host Database Group PTF level and the HIPER PTF levels. To find these PTF levels from an iSeries command prompt, run the WRKPTFGRP command.
 - Note the level of the database group.
 - For V5R2M0, the database group is SF99502.
 - For V5R3M0, the database group is SF99503.
 - Note the level of the HIPER group.
 - For V5R2M0, the HIPER group is SF99519.
 - For V5R3M0, the HIPER group is SF99529.

For more information about iSeries fixes, see:

<http://www.ibm.com/servers/eserver/support/iseries/fixes/index.html>

- ▶ The version of Windows on the PC.
- ▶ The Microsoft .NET Framework version. In Windows Explorer, navigate to <Windows Folder> → **Microsoft.NET\Framework** → <version> → **mscorlib.dll**.

Right-click **mscorlib.dll** and select **Properties**. Click the **Version** tab. Note the File Version. If multiple versions of the framework are installed, note the one being used by your application. The version of .NET Framework used by your application can be modified by settings in your app.config file or machine.config file. For information about .NET versioning, visit the MSDN Web site at:

<http://msdn.microsoft.com/>

Search for the article entitled “Using Side-by-Side Execution.”

- ▶ A description or sample program describing how to reproduce the problem. If a sample program cannot be provided to reproduce the problem, collect the **cwbmptrc** trace output, a communication trace, and the host server job log. If the problem can be reproduced without connection pooling enabled, we recommend that you disable connection pooling (set `Pooling=false` in your `ConnectionString`), and make sure you close your connections when they are complete by calling the `Close()` method of your `IDB2Connection` objects.
- ▶ A **cwbmptrc** trace output. (See 4.10.3, “Use provider traces via the cwbmptrc utility” on page 167.)
- ▶ The host server joblog. The name of the host server job can be found by querying the `JobName` connection property. The `JobName` is also recorded in the **cwbmptrc** trace output file after a connection is opened successfully. You can use the `Trace=PrintJoblog` option in your `ConnectionString` to cause the joblog to be spooled when the job ends. The `Trace` option can also be used to specify additional options, for example to include debug information in the joblog. See “Trace” on page 63 for more information about using the `Trace` `ConnectionString` property. Alternatively, you can override the `ConnectionString` using the `Trace` option; for example:

```
cwbmptrc -override "Trace=PrintJoblog, StartDebug"
```

See “Overriding your `ConnectionString`” on page 168 for more information about overriding the `ConnectionString`.

- ▶ A communications trace. (See 4.10.5, “Use communication traces via the cwbcotrc utility” on page 168.)

- The file version of the following files:
 - IBM.Data.DB2.iSeries.dll, which is usually found in your Client Access directory
 - cwbdcl.dll, which is usually found in your <Windows Folder>\system32 directory

Either use Windows Explorer to navigate to the directory where the file resides or use **Start** → **Search** to search for the specific file. When you locate the file, right-click it and select **Properties**. Click the **Version** tab and record the File Version.

Note: IBM.Data.DB2.iSeries.dll also has an Assembly version, but we want the File version.

We recommend that when you gather trace information for IBM support, turn off connection pooling by adding the following to the ConnectionString:

```
Pooling=false;
```

If any Trace ConnectionString options are being used, the traces are not turned off until the connection is terminated, and with Connection Pooling, connections are normally kept around indefinitely. See “Trace” on page 63 for information about enabling server-side traces using the Trace ConnectionString option.

With Pooling set to false, your application should explicitly call Close() on the iDB2Connection object; otherwise, your server-side traces will not be completed until the object is disposed. With Pooling set to True (the default), your application must explicitly call iDB2ProviderSettings.CleanupPooledConnections() before terminating, or the traces will not be completed. See “When pooled connections stay around too long” on page 145 for information about CleanupPooledConnections.

4.11 Writing code for provider independence

If you have used older data access technologies such as ODBC or OLE DB, you are already familiar with the idea of provider independence. With those technologies, you access data from different database providers using the same object names. The only thing that differs is usually the way you connect to the database. Example 4-120 shows how to connect to an iSeries database and run a SELECT statement to read data from the database, using the IBMDA400 OLE DB provider:

Example 4-120 Reading from an iSeries database using the IBMDA400 OLE DB Provider

```
Private cn As New ADODB.Connection
Private cmd As New ADODB.Command
Private rs As New ADODB.Recordset
Private Sub Form_Load()
    cn.ConnectionString = "Provider=IBMDA400; Data Source=myiSeries;"
    cn.Open

    'Initialize the command to select records
    'from the EMPLOYEE table.
    cmd.CommandText = "select * from sampled.b.employee"

    'Point the command to our connection object
    cmd.ActiveConnection = cn

    'Create a recordset object which contains the results
    'of executing our SELECT statement
    Set rs = cmd.Execute
```

```

        'Read records from the recordset and display them
    While rs.EOF = False
        Debug.Print "Record: "
        Debug.Print "  EMPNO: " & rs.Fields("EMPNO")
        Debug.Print "  EMPNAME: " & rs.Fields("LASTNAME") & ", " & rs.Fields("FIRSTNAME") &
" " & rs.Fields("MIDINIT")
        Debug.Print "  WORKDEPT: " & rs.Fields("WORKDEPT")
        rs.MoveNext
    Wend

    rs.Close
    cn.Close
End Sub

```

If you want to run the same application, but instead of using OLE DB with IBMDBA400, you want to use the iSeries Access for Windows ODBC driver, all you have to do is change the Connection information. Example 4-121 shows the previous example rewritten to use ODBC.

Example 4-121 Reading from an iSeries database using the iSeries Access for Windows ODBC driver

```

Private cn As New ADODB.Connection
Private cmd As New ADODB.Command
Private rs As New ADODB.Recordset
Private Sub Form_Load()
    cn.Properties("Prompt") = adPromptComplete
    cn.ConnectionString = "DRIVER=Client Access ODBC Driver (32-bit);SYSTEM=myiSeries;
ExtColInfo=1; XDynamic=0;"
    cn.Open

    'Initialize the command to select records
    'from the EMPLOYEE table.
    cmd.CommandText = "select * from sampledb.employee"

    'Point the command to our connection object
    cmd.ActiveConnection = cn

    'Create a recordset object which contains the results
    'of executing our SELECT statement
    Set rs = cmd.Execute

    'Read records from the recordset and display them
    While rs.EOF = False
        Debug.Print "Record: "
        Debug.Print "  EMPNO: " & rs.Fields("EMPNO")
        Debug.Print "  EMPNAME: " & rs.Fields("LASTNAME") & ", " & rs.Fields("FIRSTNAME") &
" " & rs.Fields("MIDINIT")
        Debug.Print "  WORKDEPT: " & rs.Fields("WORKDEPT")
        rs.MoveNext
    Wend

    rs.Close
    cn.Close
End Sub

```

While provider independence is fairly easy to implement with ADO and OLE DB, it is more difficult with ADO.NET.

With the first few versions of ADO.NET, Microsoft does not give users a good way to easily write code for provider independence. Instead of using generic object or class names,

programmers are expected to write to a specific provider using each provider's class names. For example, to open a connection and run a command using the IBM.Data.DB2.iSeries provider, use the code shown in Example 4-122.

Example 4-122 IBM.Data.DB2.iSeries provider-specific code

```
static void iSeriesProviderExample()
{
    // Create a connection to the iSeries using the
    // IBM.Data.DB2.iSeries provider
    iDB2Connection cn = new iDB2Connection("DataSource=myiSeries;
DefaultCollection=sampled;");

    // Create a command to execute on the iSeries
    iDB2Command cmd = cn.CreateCommand();
    cmd.CommandText = "select * from employee";

    // Open the connection
    cn.Open();

    // Execute the command, and get a DataReader in return.
    iDB2DataReader dr = cmd.ExecuteReader();

    // Process the result set... code not shown

    // Close the DataReader and the connection
    dr.Close();
    cmd.Dispose();
    cn.Close();
}
```

Alternatively, to perform the same task using the System.Data.SqlClient provider to talk to a SQL Server instance, you would use the code shown in Example 4-123.

Example 4-123 System.Data.SqlClient provider-specific code

```
static void SQLProviderExample()
{
    // Create a connection to a SQL Server instance using the
    // System.Data.SqlClient provider
    SqlConnection cn = new SqlConnection("Data Source=mySqlServer; User id=myUserId;
Password=myPassword; Initial Catalog=sampled;");

    // Create a command to execute on the SQL server
    SqlCommand cmd = cn.CreateCommand();
    cmd.CommandText = "select * from employee";

    // Open the connection
    cn.Open();

    // Execute the command, and get a DataReader in return.
    SqlDataReader dr = cmd.ExecuteReader();

    // Process the result set... code not shown

    // Close the DataReader and the connection
    dr.Close();
    cmd.Dispose();
    cn.Close();
}
```

If you want the ability to switch between the two different providers, keep in mind that the class names for each provider are different (iDB2Connection versus SqlConnection), so writing independent code can be difficult. In the next section we show how to make writing provider-independent code a little easier.

4.11.1 Writing provider-independent code with ADO.NET 1.0 and 1.1

The key to writing provider-independent code with early versions of ADO.NET is to take advantage of the .NET interfaces implemented by each provider. For example, because all ADO.NET providers must support a connection class that implements IDbConnection and a command class that implements IDbCommand, you can write code that uses the interface names instead of the class names. Because the .NET interface definitions do not offer a way to create a generic connection object, creating and initializing a connection object is unique for each provider. Example 4-124 shows how to write provider-independent code to read data from the database as in our previous examples. The only difference when using the two providers is the way you create and initialize your connection.

Example 4-124 Provider-independent code using ADO.NET 1.0 and 1.1 interfaces

```
static void GenericProviderExample(String provider)
{
    IDbConnection cn;

    // Create a connection to either the iSeries, or to
    // a SQL Server instance, depending on what was
    // passed as input.
    if (provider.CompareTo("iseries") == 0)
    {
        // Create a connection to the iSeries using the
        // IBM.Data.DB2.iSeries provider
        cn = new iDB2Connection("DataSource=myiSeries; DefaultCollection=sampledb;");
    }
    else
    {
        // Create a connection to a SQL Server instance using the
        // System.Data.SqlClient provider
        cn = new SqlConnection("Data Source=mySqlServer; User id=myUserId;
Password=myPassword; Initial Catalog=sampledb;");
    }

    // Create a command to execute
    IDbCommand cmd = cn.CreateCommand();
    cmd.CommandText = "select * from employee";

    // Open the connection
    cn.Open();

    // Execute the command, and get a DataReader in return.
    IDataReader dr = cmd.ExecuteReader();

    // Process the result set... code not shown

    // Close the DataReader and the connection
    dr.Close();
    cmd.Dispose();
    cn.Close();
}
```

This is a simple example, but it shows how you can write provider-independent code with just a little work. You can use provider-independent parameters and transactions just as easily, for example:

```
cmd.Parameters.Add(cmd.CreateParameter());  
IDataParameter p = (IDataParameter)cmd.Parameters[0];  
IDbTransaction t = cn.BeginTransaction();
```

Although this works well for many of the ADO.NET objects, this approach has some drawbacks. Because some ADO.NET classes must be *newed* in order to get an instance (namely Connection, DataAdapter, and CommandBuilder), you still have to write provider-specific code to create them, as shown with the Connection object in Example 4-124 on page 174.

4.11.2 Writing provider-independent code with ADO.NET 2.0

Writing code for provider independence is a little tricky with earlier versions of ADO.NET, so Microsoft provides an easier solution in their ADO.NET Version 2.0. They defined a set of base classes that data providers can inherit from; for example, base classes DbConnection and DbCommand are used for connections and commands. In addition, they have defined the notion of provider “factory classes” that can be used to create provider-specific objects. You can read more about ADO.NET 2.0 at the MSDN Library Web site:

<http://msdn.microsoft.com/library/>

Select **.NET Development** → **Data Access and Storage** → **ADO.NET** → **Technical Articles** → **ADO.NET 2.0**.

As of this writing, the IBM.Data.DB2.iSeries provider does not support the ADO.NET 2.0 base classes and provider factory classes. If or when this support is added in the future, applications will be able to write code more easily that works with different .NET providers.



IBM DB2 for LUW .NET provider

This chapter describes the IBM DB2 for LUW .NET provider, integrated in DB2 UDB for Linux, UNIX, and Windows (codenamed “Stinger”) and in DB2 Connect, the IBM product that provides functionality for accessing data stored in DB2 UDB for iSeries as well as DB2 for z/OS® and OS/390®. DB2 Connect offers capabilities to access DB2 family members by way of several SQL interfaces, gateway functions such as connection pooling, and federated database functionality. Many applications that support the DB2 family of products leverage DB2 Connect as a key middleware component.

In this chapter, we see how to access a DB2 Universal Database Version 5, Release 1 (or later) for AS/400 and iSeries, through DB2 Connect. The IBM DB2 for LUW .NET provider is also referred to by the namespace it defines, *IBM.Data.DB2*.

5.1 DB2 Connect overview

For DB2 clients on a LAN, a DB2 Connect server enables access to data that is stored on host or iSeries systems. DB2 Universal Database Enterprise Server Edition includes the DB2 Connect Server Support component. All references to DB2 Connect Enterprise Edition also apply to the DB2 Connect Server Support component.

DB2 Connect provides transparent access to host or iSeries data through a standard architecture for managing distributed data. This standard is known as Distributed Relational Database Architecture (DRDA). DRDA enables applications to establish a fast connection to host and iSeries databases without expensive host or iSeries components or proprietary gateways.

Although DB2 Connect is often installed on an intermediate server machine to connect DB2 clients to a host or iSeries database, it is also installed on machines where multiple local users want to access the host or iSeries servers directly. For example, DB2 Connect may be installed on a large machine with many local users.

DB2 Connect may also be installed on a Web server, Transaction Processor (TP) monitor, or other three-tier application server machines with multiple local SQL application processes and threads. In these cases, you can choose to install DB2 Connect on the same machine for simplicity or on a separate machine to offload CPU cycles.

5.2 Installing and configuring DB2 Connect

This section describes how to install DB2 Connect Enterprise Edition on Windows operating systems. Before using the DB2 Connect Install program to install the DB2 .NET Data Provider, the .NET Framework V1.1 must be installed on the computer; otherwise the DB2 Install program will not install the DB2 .NET Data Provider.

For information about the supported AS/400 and iSeries versions, and the required iSeries server PTFs, see information APAR II13348. You can view information APARs by accessing the IBM eServer iSeries Support Web site at:

<http://techsupport.rchland.ibm.com>

Select **Technical Databases** → **Authorized Problem Analysis Reports APARs** → **Search APARs**. Type your APAR number, in this case II13348. Then select the APAR II13348.

Only the .NET Framework Version 1.1 and Visual Studio .NET 2003 are supported for use with DB2 for VSE & VM, and DB2 for iSeries servers. The .NET Framework Version 1.0 and Visual Studio .NET 2002 are not supported for use with these servers.

Note: There are several different versions of DB2 Connect, from a Personal Edition to an edition with unlimited access. For more information, visit the DB2 Universal Database for Linux, UNIX, and Windows Web site at:

<http://www.ibm.com/software/data/db2/udb/>

5.2.1 Host server jobs

When you run applications that use the DB2 for LUW .NET provider, much of the work is performed by the iSeries server on behalf of your application. This is accomplished with the help of host server jobs that run on the iSeries. The provider handles transferring the commands and data back and forth between your PC and the host server jobs through the

DRDA interface. The host server has a prestart job, QRWTSRVR, which normally runs under the QUSRWRK subsystem. The host servers are installed as part of the operating system, 5722SS1 option 12 (Host Servers). Before you can use the DB2 for LUW .NET provider to communicate with your iSeries host server jobs, the server jobs must be active. For more information about the DRDA interface and associated server jobs, go the iSeries Information Center at:

<http://www.iseries.ibm.com/infocenter>

Select **Information center home** → **Printable PDFs and manuals** → **Distributed Database Programming**.

5.2.2 Prerequisites

Before you launch the DB2 Setup wizard, ensure that your system meets:

- ▶ Memory requirements
- ▶ Hardware, distribution, and software requirements
- ▶ Disk requirements

If you are planning to use LDAP on Windows 2000 or Windows Server 2003 (32-bit and 64-bit), you must extend the directory schema.

It is recommended that you use an Administrator account to perform the installation. The Administrator account must belong to the local administrator's group on the Windows computer where you are installing your DB2 product and should have the following advanced user rights:

- ▶ Act as part of the operating system
- ▶ Create token object
- ▶ Increase quotas
- ▶ Replace a process level token

You can perform the installation without advanced user rights, but the setup program may be unable to validate accounts.

5.2.3 Installation procedure

To install DB2 Connect Enterprise Edition:

- ▶ Log on to the system as a user with administrator authority.
- ▶ Close all programs so the installation program can update files as required.
- ▶ Insert the CD-ROM into the drive. The auto-run feature automatically starts the DB2 Setup wizard. The DB2 Setup wizard determines the system language and launches the setup program for that language. If you want to run the setup program in a different language, or the setup program fails to auto-start, you can run the DB2 Setup wizard manually.

The DB2 Launchpad opens as shown in Figure 5-1 on page 180.

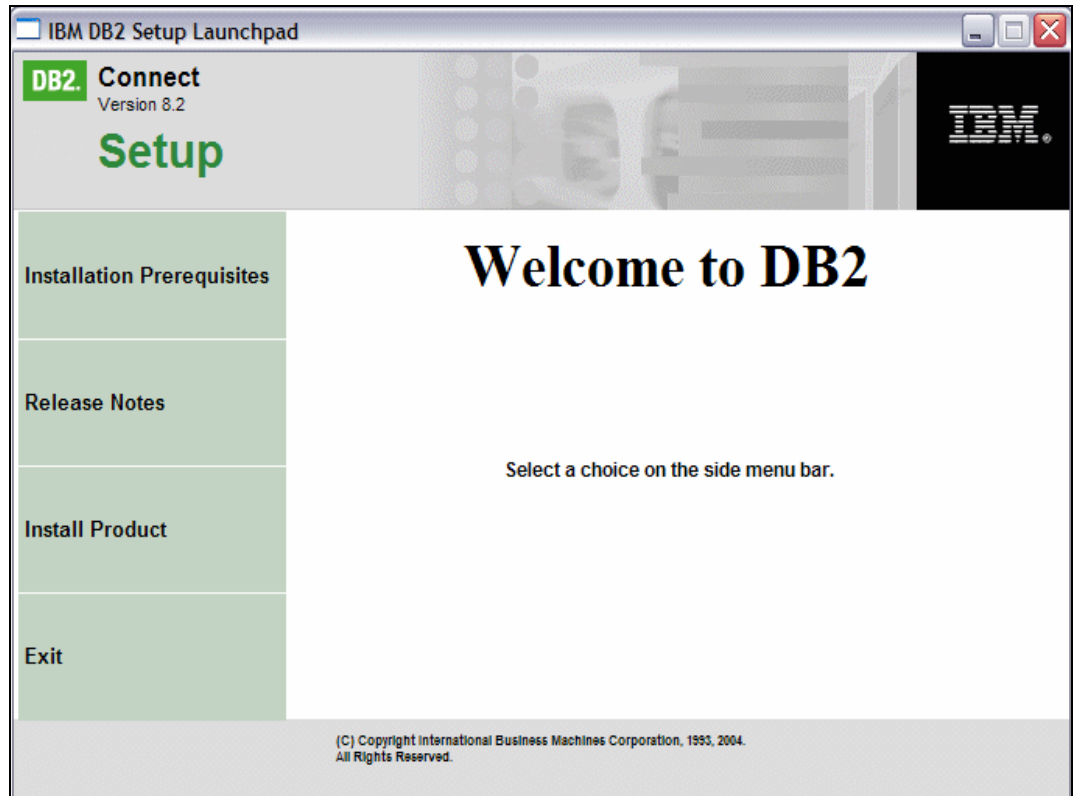


Figure 5-1 DB2 Connect Enterprise Edition setup

From this window, you can view the installation prerequisites and the release notes, or you can proceed directly to the installation.

When you have begun the installation, proceed by following the setup program's prompts. Online help is available to guide you through the remaining steps; either click Help or press F1. You can click Cancel at any time to exit the installation.

For information about errors encountered during installation, see the db2.log file, which stores general information and error messages resulting from the install and uninstall activities. By default, the db2.log file is located in the x:\db2log directory, where x: represents the drive on which your operating system is installed.

5.2.4 Connecting to an iSeries database

DB2 Connect enables you to connect to your iSeries database in several ways, including:

- ▶ Specify the connection information directly in your `ConnectionString`.
- ▶ Configure a connection using the Configuration Assistant, then use that information in your `ConnectionString`.

Connecting using a simple `ConnectionString`

The easiest way to connect to your iSeries using the DB2 for LUW .NET provider is to specify your connection information in the `ConnectionString`. See 5.6.5, "Using the `DB2Connection` object and the `ConnectionString`" on page 199 for a description of all values you can specify in your `ConnectionString`. Example 5-1 on page 181 shows an example of a simple `ConnectionString`.

Example 5-1 A simple ConnectionString

```
//Initialize the connection string
String myConnString = "Server=myiSeries:446; Database=myRDB; UID=myUserid;
PWD=myPassword;";

//Create a connection object
DB2Connection cn = new DB2Connection(myConnString);

//Connect to the iSeries
cn.Open();
```

In this example:

- ▶ Server is the name or IP address of your iSeries server.
- ▶ 443 is the standard port number for DRDA on the iSeries.
- ▶ Database is the name of your relational database (RDB) on the iSeries.
- ▶ UID is your iSeries user ID.
- ▶ PWD is your iSeries password.

Hint: You can determine the RDB (database) name for your iSeries by using the WRKRDBDIRE command. The system database is indicated by a Remote Location of *LOCAL.

Connecting using the Configuration Assistant

There are times when you may want to use the Configuration Assistant to configure your database connection. Follow these steps to configure a connection to an iSeries database using the Configuration Assistant:

1. Launch the Configuration Assistant from your PC desktop by selecting **Start → Programs → IBM DB2 → Set-up Tools → Configuration Assistant**.
2. Click **Add** to add a new data source and respond **Yes** to launch the Add Database Wizard.
3. In the window Select how you want to set up a connection, click **Manually configure a connection to a database**.
4. Click the **Protocol** tab, and select **TCP/IP** for protocol and the option **Connect directly to the server**. Click **Next**.
5. In the window Specify TCP/IP communication parameters, type the name of your iSeries host in the Host name field. The Port number should be 446 to connect to the DRDA server. Click **Next**.
6. In the window Specify the name of the database to which you want to connect, type the name of your iSeries database (RDB name). The database alias is automatically filled in with the same value as your database name. Click **Next**.
7. In the window Register this database as a data source, click **Next**.
8. In the window Specify the node options, select **OS/400** for the operating system. Click **Finish**.

When your connection has been configured through the Configuration Assistant, you can connect as shown in Example 5-2.

Example 5-2 A ConnectionString that uses a pre-configured database name

```
//Initialize the connection string
String myConnString = "Database=myDatabase; UID=myUserid; PWD=myPassword;";

//Create a connection object
```

```
DB2Connection cn = new DB2Connection(myConnString);

//Connect to the iSeries
cn.Open();
```

In this example:

- ▶ Database is the name of the database alias you configured in step 6 on page 181.
- ▶ UID is your iSeries user ID.
- ▶ PWD is your iSeries password.

5.3 IBM DB2 Development Add-In overview

IBM DB2 Add-In is the collection of features that integrate with the Microsoft Visual Studio .NET development environment. The Add-In has the following features, through which you can perform operations on DB2:

- ▶ Launch various DB2 development and administration tools.
- ▶ Create and manage DB2 projects in the Solution Explorer.
- ▶ Access and manage DB2 data connections in the IBM Explorer.
- ▶ Create and modify DB2 scripts, including scripts to create stored procedures and user-defined functions (UDFs).

The DB2 Development Add-In for Visual Studio .NET extends the DB2 server support to include:

- ▶ DB2 for z/OS® and OS/390® versions 6, 7, and 8
- ▶ DB2 for iSeries® (AS/400®) versions 5.1 and 5.2

Note: Although it may work, Version 8.2 of Development Add-In for Visual Studio .NET was not tested using DB2 for iSeries Version 5.3.

- ▶ DB2 for Linux, UNIX®, and Windows V8.1 and later

Currently not all of the features are available for the various DB2 servers. Table 5-1 lists the restricted features.

Table 5-1 Features availability by distribution

Restricted feature	Distributed	iSeries	z/OS V6	z/OS V7	z/OS V8
Generate table/view create script	Yes	No	No	No	No
Create scalar functions	Yes	Yes	No	Yes	Yes
Create table functions	Yes	No	No	No	No
Generate function create script	Yes	Yes	No	No	No
Generate Web service	Yes	No	No	No	No
Tablespaces for create table	Yes	No	Yes	Yes	Yes

5.3.1 Registering the IBM DB2 Development Add-In

The IBM DB2 Development Add-In is installed on the client as a part of IBM DB2 Universal Database for Windows, Version 8.x. You can check whether the Add-In is installed on the client PC by examining the splash screen at Visual Studio .NET startup.

The DB2 Add-In can be registered in different ways:

1. Automatic installation

If you installed Visual Studio .NET before installing DB2, the DB2 Development Add-In is registered automatically. If you installed Visual Studio .NET after you installed DB2 UDB or you modified your installation, you must manually register the Add-In.

2. By using the IBM DB2 Set-up Tools menu

The IBM Add-In can be registered by selecting Windows **Start** → **Programs** → **IBM DB2** → **Set-up Tools** → **Register Visual Studio Add-Ins** menu as shown in Figure 5-2.

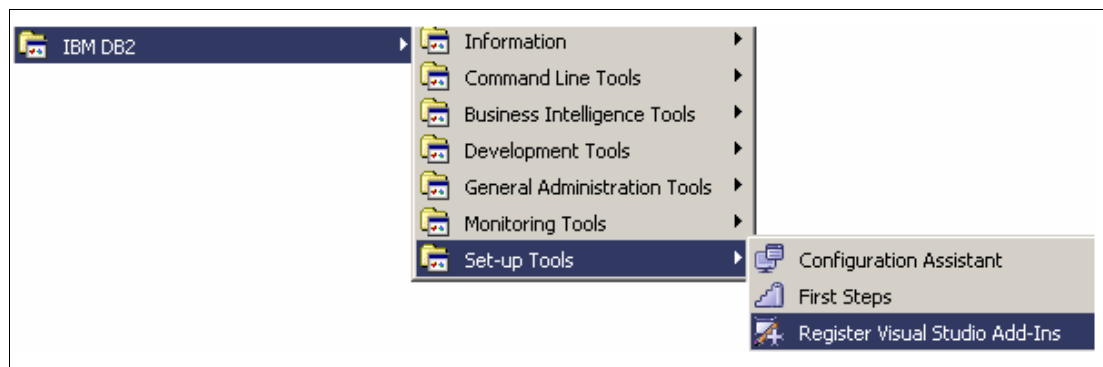


Figure 5-2 Register Visual Studio Add-Ins menu

3. Register manually

You can also register IBM DB2 Development Add-In with Visual Studio .NET by performing the following steps:

- a. Exit Visual Studio .NET.
- b. In the SQLLIB/bin directory, run the db2vsrgx.bat file.
- c. Start Visual Studio .NET.
- d. The IBM DB2 Development Add-In tools and help are registered.

5.3.2 Unregistering the IBM DB2 Development Add-In

To unregister the IBM DB2 Development Add-In, perform the following steps:

- a. Exit Visual Studio .NET.
- b. In SQLLIB/bin, run db2vsrgx.bat.

The IBM DB2 Development Add-In tools and help are unregistered. You can unregister the IBM DB2 Development Add-In help only by entering the following command in your command window:

```
db2vsreg -unregister doc
```

Note: The IBM DB2 Development Add-In is automatically uninstalled when you uninstall the IBM DB2 Universal Database server or client products.

5.3.3 DB2 Toolbar

With the DB2 Toolbar, you can launch the following DB2 tools:

- ▶ Development Center
- ▶ Control Center
- ▶ Replication Center
- ▶ Command Editor
- ▶ Task Center
- ▶ Health Center
- ▶ Journal
- ▶ Information Center

Figure 5-3 shows how it is depicted in the Microsoft Development Environment.

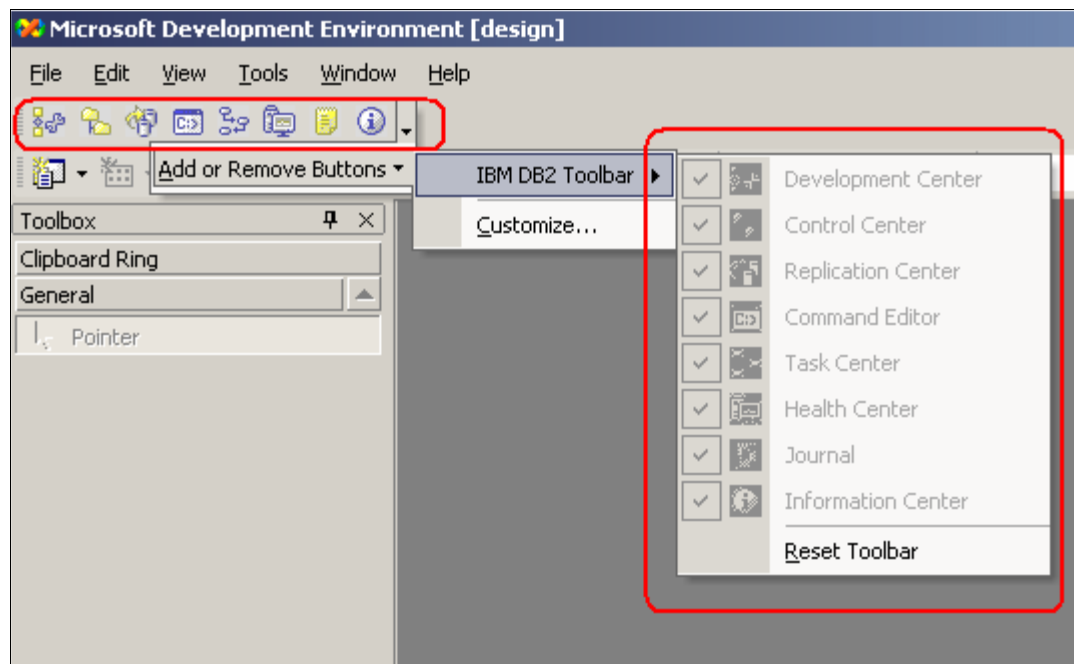


Figure 5-3 IBM DB2 Toolbar

5.3.4 DB2 Database Project type

The IBM DB2 Development Add-In introduces an IBM Projects folder, which includes a DB2 Database Project type for developing DB2 database server scripts (Figure 5-4 on page 185).

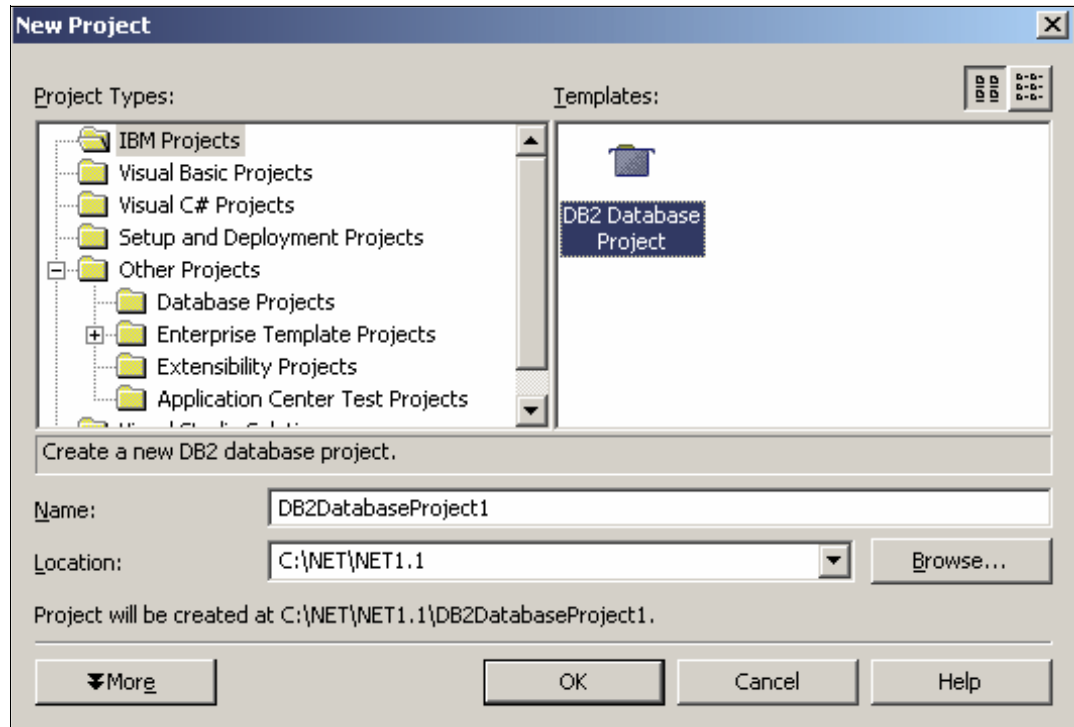


Figure 5-4 DB2 Database Project

With a DB2 Database Project, you can:

- ▶ Add new or existing SQL stored procedure scripts.
- ▶ Add new or existing CLR stored procedure scripts.
- ▶ Add new or existing SQL UDF scripts.
- ▶ Add new or existing scripts based on generic templates.
- ▶ Add new or existing SQL table, index, view, and triggers scripts.
- ▶ Specify build configuration options, including script build order.
- ▶ Check script files into Microsoft Visual Source Safe source-control management system.

The *Solution Explorer* changes when you choose a DB2 Database project (Figure 5-5).

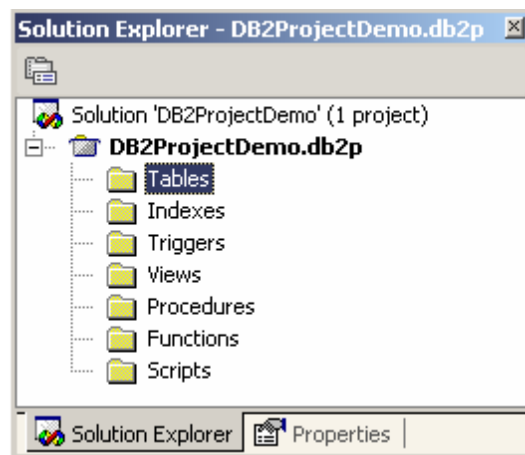


Figure 5-5 DB2 project Solution Explorer

5.3.5 IBM Explorer

The IBM DB2 Development Add-In extends the Visual Studio .NET environment by adding a new tool called IBM Explorer (Figure 5-6), a dockable window that is similar to the Visual Studio .NET Server Explorer.

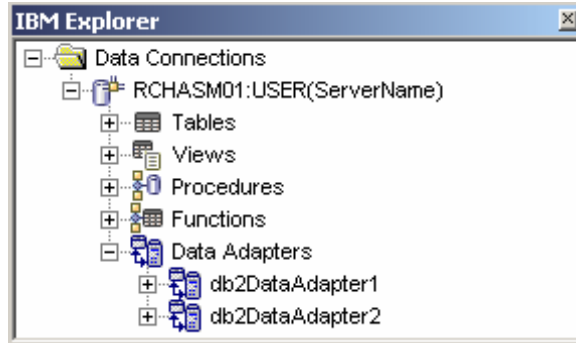


Figure 5-6 IBM Explorer

IBM Explorer provides Visual Studio .NET users with access to IBM database connections using the Data Connections folder, which is designed specifically for DB2 managed provider connections. From the Data Connections folder in the IBM Explorer, you can:

- ▶ Access information about tables and views, including columns, triggers, and index details.
- ▶ Work with multiple named DB2 connections supporting connect on demand technology.
- ▶ Specify database catalog filters and local caching for higher performance and scalability.
- ▶ View properties of server objects including tables, views, stored procedures, and functions.
- ▶ View source code for DB2 stored procedures and functions.
- ▶ Retrieve and update data from tables and views.
- ▶ Execute test runs for stored procedures and UDFs.
- ▶ View source code for stored procedures and functions.
- ▶ Generate ADO .NET code.
- ▶ Test DB2 procedures and functions.
- ▶ Create and manage reusable data adapters.
- ▶ Generate DB2 embedded application server Web services.

5.4 IBM DB2 data controls

In .NET, you can generate Windows front-end or Win Form code for interacting with a database in two ways: either by writing code for each functionality as we have seen in previous sections or by simply dragging and dropping (binding) database-related controls. Dragging and dropping controls for database interaction requires few lines of code, and therefore provides an easy and fast way to develop GUI applications.

Data controls that are available in DB2 are:

- ▶ DB2Connection
- ▶ DB2Command
- ▶ DB2DataAdapter

Figure 5-7 shows how it is depicted in the toolbox of Microsoft Development Environment.

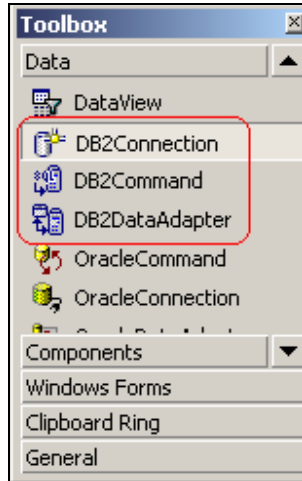


Figure 5-7 DB2 Data objects in Toolbox

To understand how to use each DB2 data control, we offer an example. Data controls can be used in several combinations; for example, you can use DB2Connection and DB2Command to execute command text, and programmatically you can assign the result to a DB2DataReader. In our demo example, we use the DB2Adapter object to get the result data from the EMPLOYEE table and then display it in a DataGrid control.

To use Data Controls on a Windows form (Win Form):

1. Create a new Windows application in C#.
2. Drag a **DB2Connection** control from the toolbox to your Windows form. Specify the ConnectionString property of DB2Connection control in Property Explorer as:

```
server=myDB2:446;database=myDatabase;Connect Timeout=30;user Id=myUID;Persist  
Security Info=true;password=myPWD;
```

Note: Be sure to add the Persist Security Info=true parameter in the ConnectionString property. This parameter preserves password information needed to populate a DataGrid.

3. Drag a **DB2DataAdapter** control from the toolbox to your Windows form. As soon as you drop the DB2DataAdapter control onto the form, a DB2 Data Adapter Configuration Wizard opens, as shown Figure 5-8 on page 188.

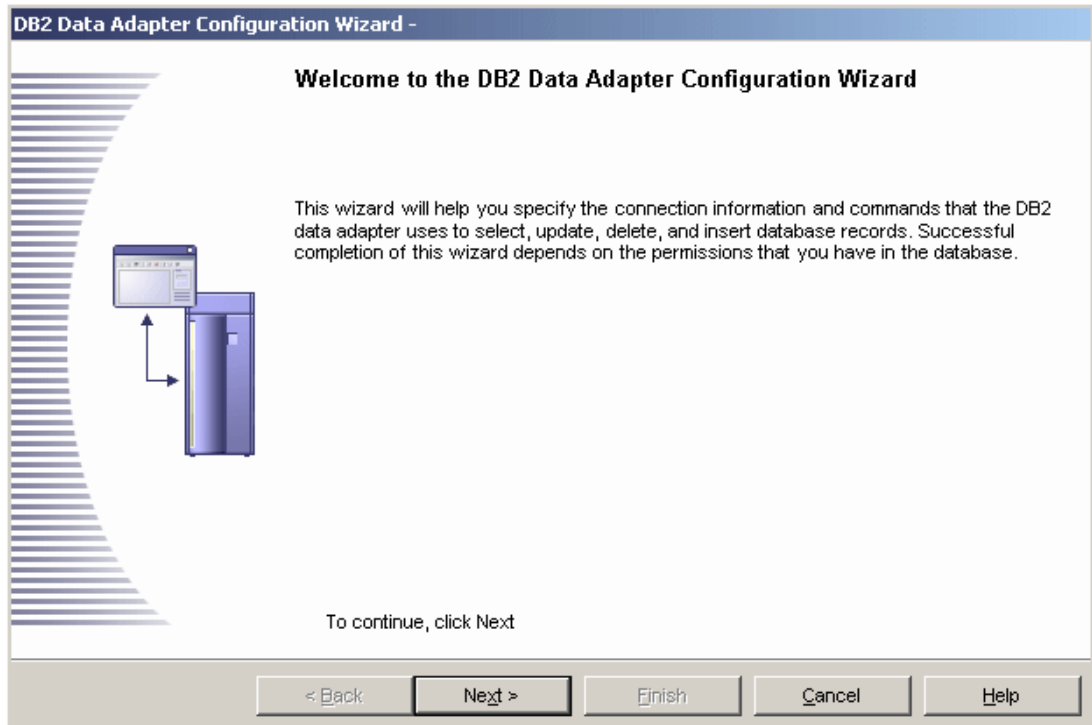


Figure 5-8 DB2 Data Adapter Configuration Wizard

4. Click **Next** to open step 1 for specifying connection details, as shown in Figure 5-9. If you have already created a connection in IBM Explorer, connection details appear automatically in the Connection Name box. Otherwise, you can specify connection details by clicking the New Connection button. Click **Next** to go to step 2.

Figure 5-9 DB2 Data Connection

5. In SQL Statement Options, you can specify whether you want to perform INSERT, UPDATE or DELETE operations on your data source. By selecting the options, the wizard automatically creates the necessary actions.

The default SQL type in the wizard is SELECT. In our example in Figure 5-10, we are not performing an UPDATE operation on the data source, so none of the options is selected. Click **Next** to go to step 3.

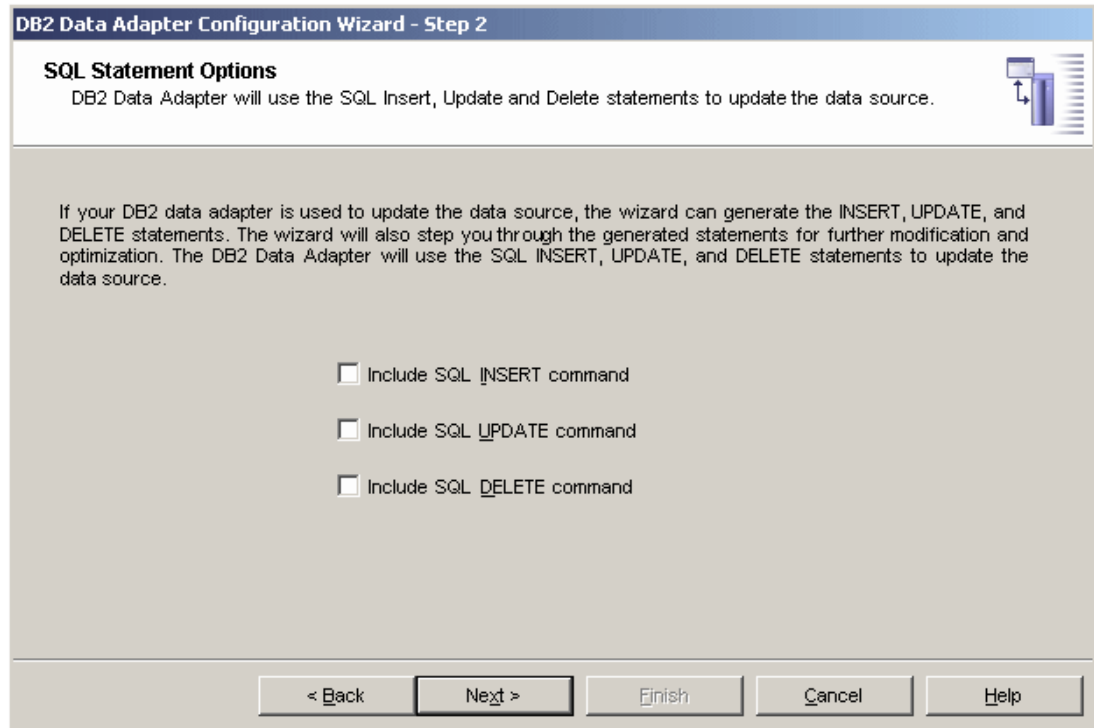


Figure 5-10 DB2 SQL statement options

6. Specify a SELECT statement or Stored procedure. Enter a SQL statement to select all employees from the EMPLOYEE table, as shown in Figure 5-11. The SQL editor supports SQL syntax colorization and intellisense to select database objects such as table name.

This step also enables you to specify parameters and table mappings, and provides functionality to test your edited SQL statement.

Click **Next**.

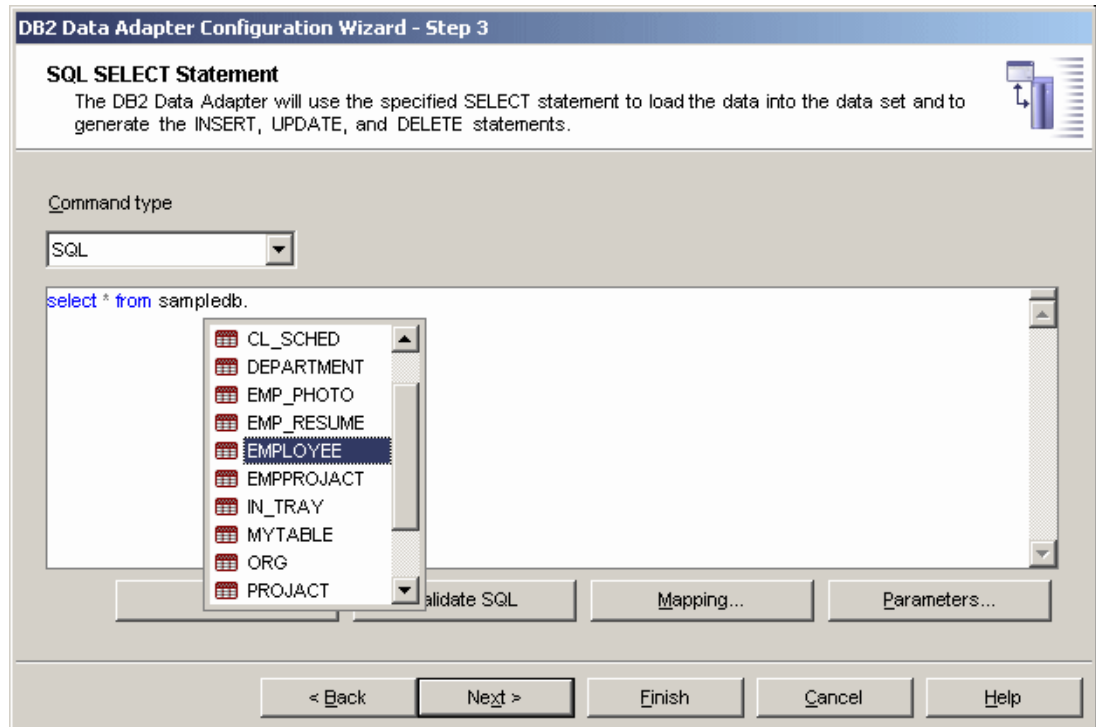
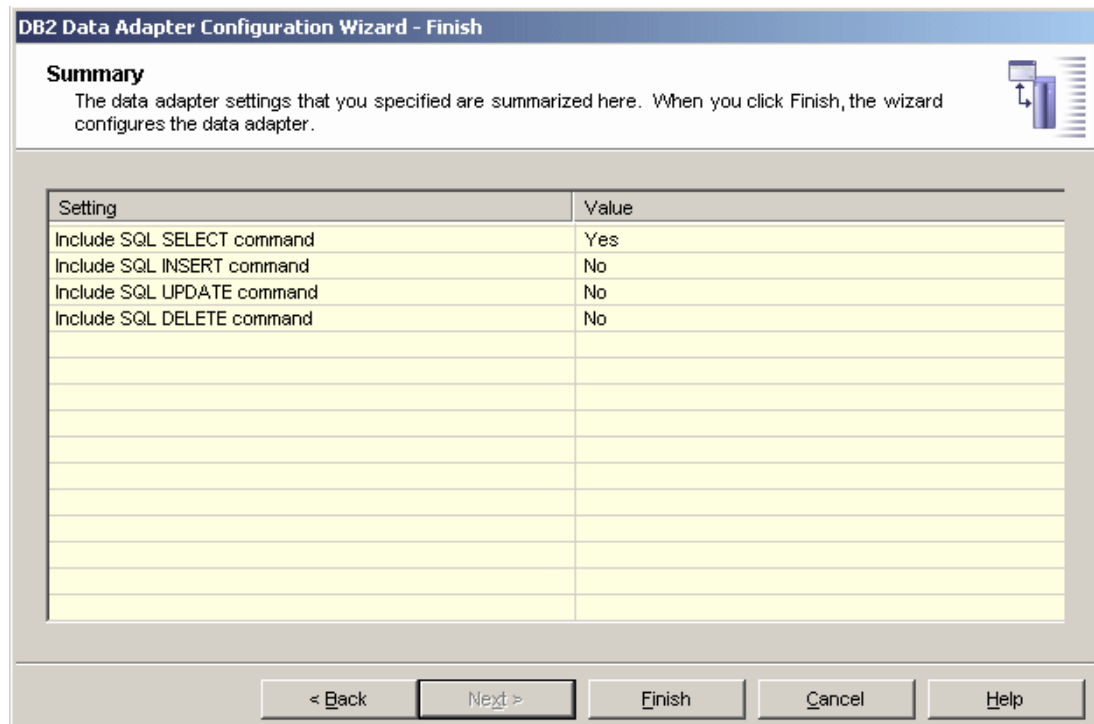


Figure 5-11 SQL editor: intellisense feature

7. The Summary window displays a summary of all SQL statements you have selected and edited, as shown in Figure 5-12. Click **Finish** to complete the wizard steps.



DB2 Data Adapter Configuration Wizard - Finish

Summary
The data adapter settings that you specified are summarized here. When you click Finish, the wizard configures the data adapter.

Setting	Value
Include SQL SELECT command	Yes
Include SQL INSERT command	No
Include SQL UPDATE command	No
Include SQL DELETE command	No

< Back Next > Finish Cancel Help

Figure 5-12 Wizard Summary

8. To bind DataAdapter controls, generate a data set: Right-click the newly configured **db2DataAdapter1** control and select **Generate Data Set**, as shown in Figure 5-13.

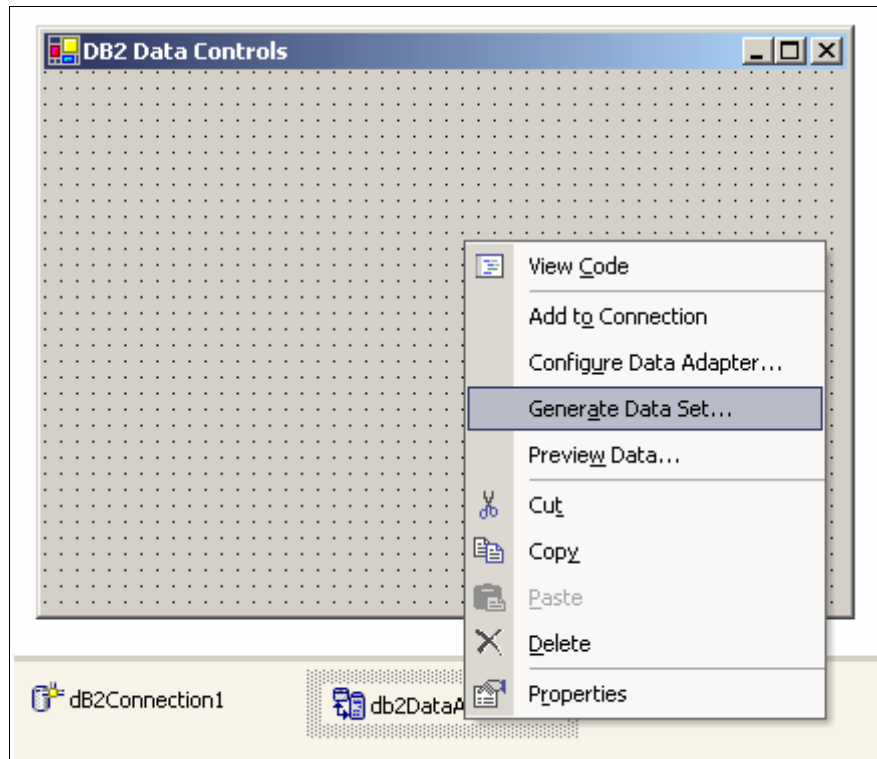


Figure 5-13 Generating Data Set from Data Adapter

9. This opens a Generate Data Set window (Figure 5-14). The Generate Data Set dialog provides functionality to select or specify a Data Set object or to select table names. Click **OK** to generate a db2DataSet1 object at the bottom of your design window.

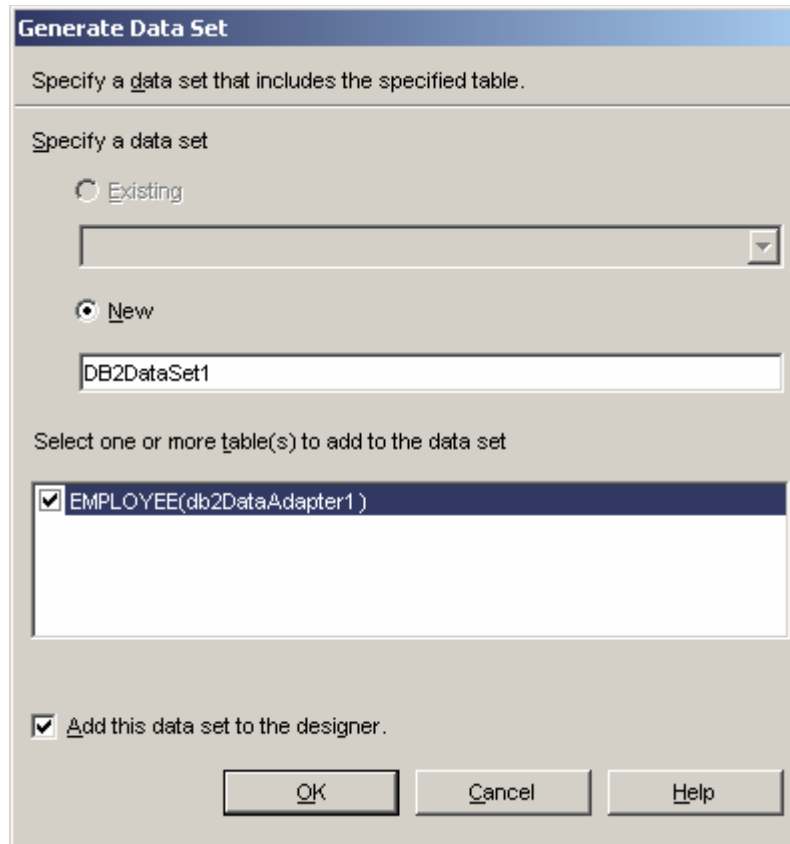


Figure 5-14 Configuring Data Set

10. To display the data from the generated data set in the DataGrid, drag a DataGrid control and a command button. Rename the command button by changing its text property to Fill DataGrid, then double-click it to open a code window. In the code window, write the code shown in Example 5-3 to populate the DataGrid control.

Example 5-3 Configuring button click action

```
private void button1_Click(object sender, System.EventArgs e)
{
    dataGrid1.DataSource = db2DataSet11;
    db2DataAdapter1.Fill (db2DataSet11);
}
```

Figure 5-15 shows a snippet of the demo example that appears after executing the application when you click Fill DataGrid.

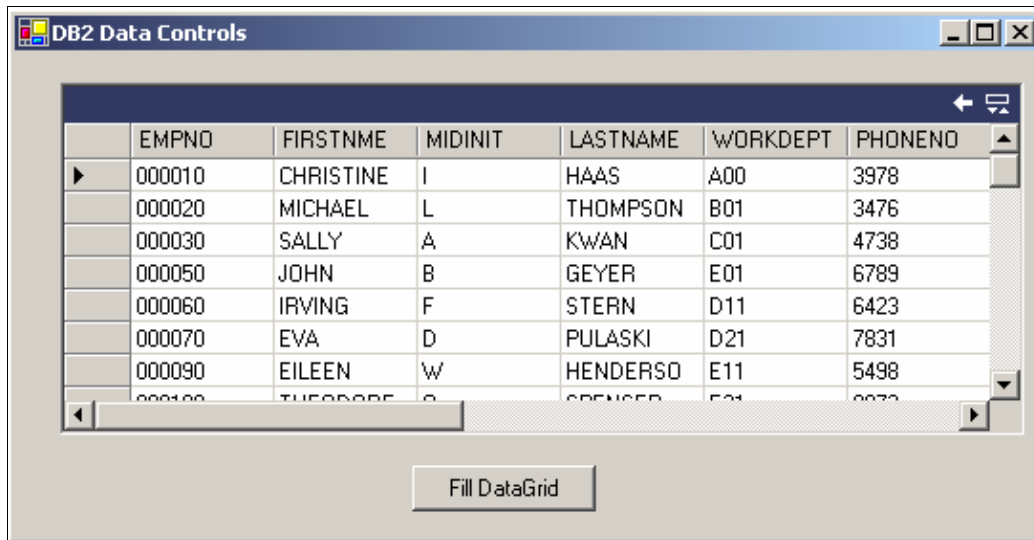


Figure 5-15 Displaying data using data controls

5.5 LUW provider features

This section describes how you can use the LUW provider and discusses the features that are supported and those not supported yet.

5.5.1 Classes to implement ADO.NET interfaces

Figure 5-16 shows the DB2 LUW provider object model.

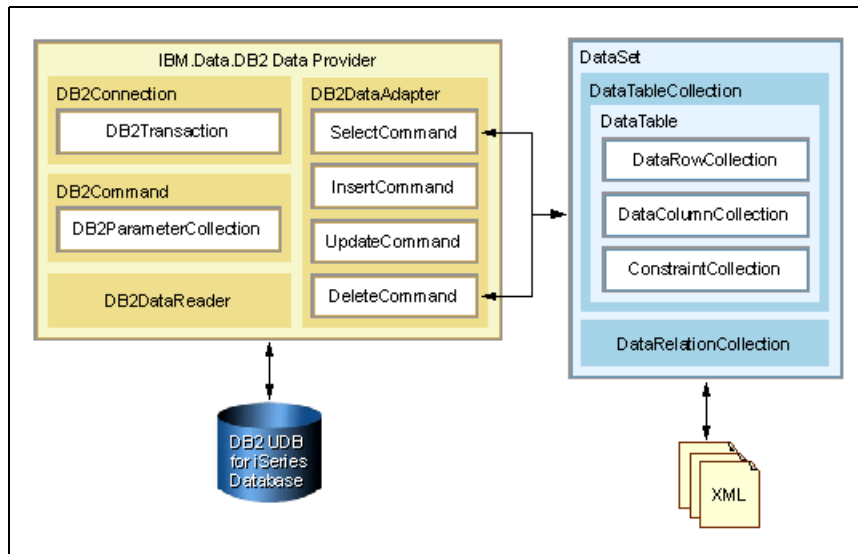


Figure 5-16 DB2 LUW provider object model

Table 5-2 shows the classes included in the LUW provider.

Table 5-2 Classes implemented in LUW provider

Class	Description
DB2Command	Represents a SQL statement or stored procedure to execute against a database.
DB2CommandBuilder	Automatically generates single-table commands that are used to reconcile changes made to a DataSet with the associated database.
DB2Connection	Represents an open connection to a database.
DB2DataAdapter	Represents a set of data commands and a connection to a database that are used to fill the DataSet and update the database.
DB2DataReader	Provides a way to read a forward-only stream of data rows from a database.
DB2Error	Collects information relevant to a warning or error returned by the database.
DB2ErrorCollection	Collects all errors generated by the DB2DataAdapter.
DB2Exception	The exception that is generated when an error is returned by an DB2 database.
DB2InfoMessageEventArgs	Provides data for the InfoMessage event.
DB2Parameter	Represents a parameter to a DB2Command and, optionally, its mapping to a DataColumn.
DB2ParameterCollection	Represents a collection of parameters relevant to an DB2Command as well as their respective mappings to columns in a DataSet.
DB2Permission	Enables the DB2 .NET Data Provider to ensure that a user has a security level adequate to access a DB2 database.
DB2PermissionAttribute	Associates a security action with a custom security attribute.
DB2RowUpdatedEventArgs	Provides data for the RowUpdated event.
DB2RowUpdatingEventArgs	Provides data for the RowUpdating event.
DB2Transaction	Represents a SQL transaction to be made at a database.

5.5.2 Data types

The DB2 for LUW provider maps DB2 data types to .NET data types, according to Table 5-3.

Table 5-3 Mapping of DB2 data types to .NET data types

DB2 type enum	DB2 data type	.NET data type
SmallInt	SMALLINT	System.Int16
Integer	INTEGER	System.Int32
BigInt	BIGINT	System.Int64
Real	REAL	System.Single
Double	DOUBLE PRECISION	System.Double
Float	FLOAT	System.Double
Decimal	DECIMAL	System.Decimal
Numeric	DECIMAL	System.Decimal
Date	DATE	System.DateTime
Time	TIME	System.TimeSpan
TimeStamp	TIMESTAMP	System.DateTime
Char	CHAR	System.String
VarChar	VARCHAR	System.String
Binary	CHAR FOR BIT DATA	System.Byte[]
VarBinary	VARCHAR FOR BIT DATA	System.Byte[]
Graphic	GRAPHIC	System.String
VarGraphic	VARGRAPHIC	System.String
CLOB	CLOB	System.String
BLOB	BLOB	System.Byte[]
DbCLOB	DBCLOB(N)	System.String

5.5.3 Unsupported features

The LUW provider, when used to access DB2 UDB for iSeries, does not support the following features:

- ▶ User-defined data types (UDTs)
- ▶ Datalink data type
- ▶ The DB2 for LUW .NET provider does not include the ability to set certain iSeries-specific settings, such as System Naming and Library List.

5.6 Getting started

To begin using the LUW provider, you must know how to get ready to write an application using it, which we demonstrate in this section.

5.6.1 Starting Visual Studio .NET

After you have installed DB2 Connect, the splash window of Visual Studio .NET displays the icon and text corresponding to IBM DB2 Tools. This indicates that the provider is installed and the help library has been set up properly.

Most of the samples in this chapter use a Console application. To create such a project, click the **Console Application** icon and fill in the information as shown in Figure 5-17.

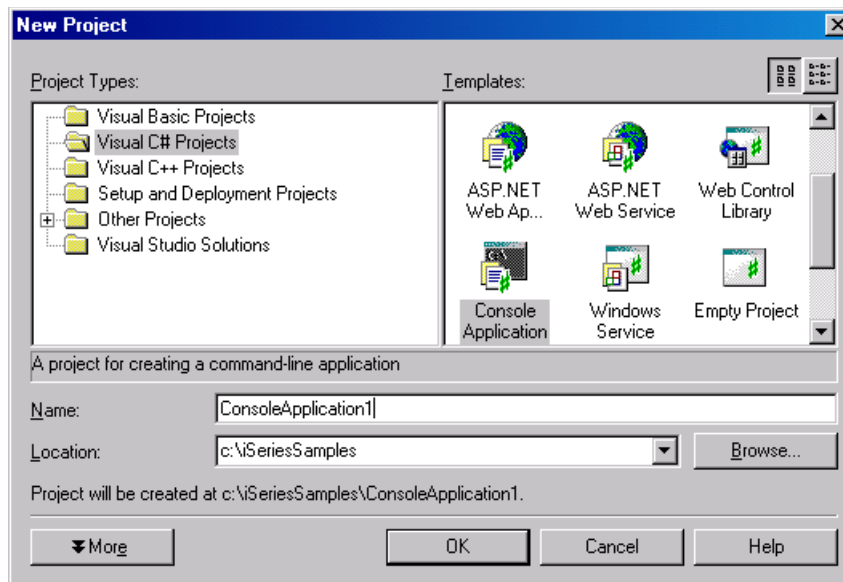


Figure 5-17 Creating a C# Console application

5.6.2 Displaying the technical reference

You can access the technical reference from Visual Studio .NET. From the Visual Studio .NET IDE, select **Help** → **Contents** to open the Contents window (Figure 5-18). Expand **IBM DB2 .NET Data Provider** to display information about classes that are implemented in this provider, including DB2 .NET Data Provider Samples.

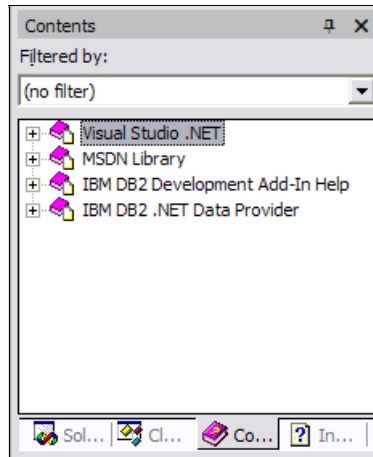


Figure 5-18 Viewing the DB2 LUW provider technical reference

5.6.3 Adding an assembly reference to the provider

To use the IBM.Data.DB2 provider classes, you must add a reference. In the Project Explorer window, right-click your project name and select **Add Reference** to open the Add Reference window shown in Figure 5-19. Click **OK** and you have now an assembly reference to the IBM.Data.DB2 provider.

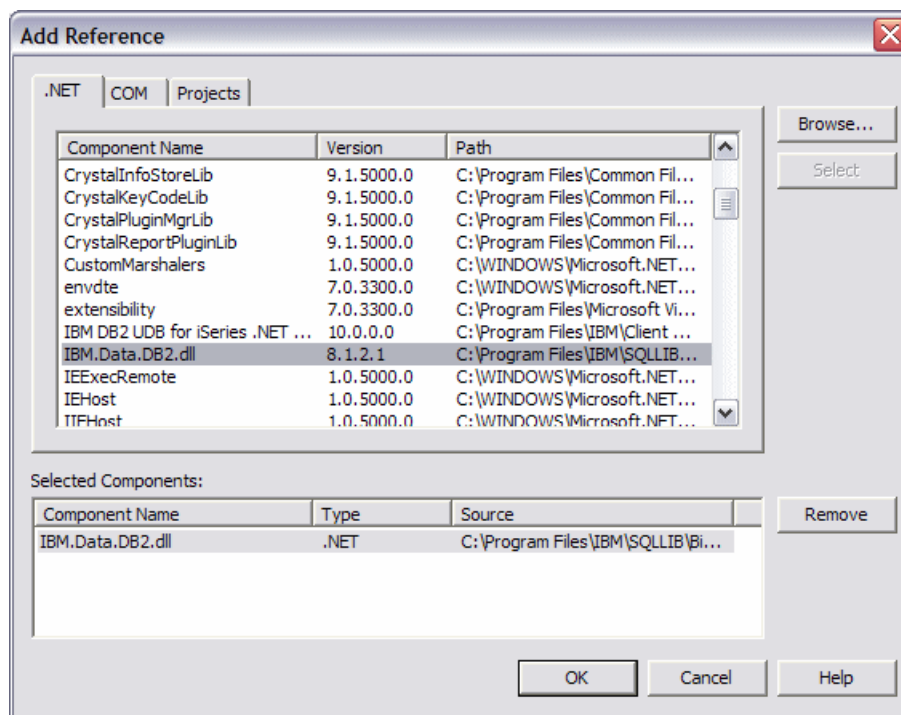


Figure 5-19 Adding a reference

5.6.4 Adding a namespace directive

A *namespace directive* imports the types contained in a namespace into the immediately enclosing compilation unit or namespace body, enabling the identifier of each type to be used without qualification. In a class, insert code to add a *using* directive as shown:

```
using IBM.Data.DB2;
```

To add a namespace directive using Visual Basic .NET, add an *Imports* statement to your Visual Basic source file:

```
Imports IBM.Data.DB2
```

5.6.5 Using the DB2Connection object and theConnectionString

In this section, we explain ConnectionString attributes in detail. Table 5-4 describes all the attributes for the ConnectionString for the DB2 LUW provider.

Table 5-4 ConnectionString attributes for the DB2 LUW provider

Attribute	Description	Default value (if omitted)
Database	Database alias (for catalogued database) or RDB name	
User ID; UID	User ID	
Password; PWD	Password	
Server	Server name with optional port number for direct connection (<server name/IP address>[:<port>])	
CurrentSchema	The schema to be used after a successful connection. Upon a successful connection, a SET CURRENT SCHEMA statement is sent to the DB2 server. This enables the application to name SQL objects without having to qualify them by a schema name.	
Connection Lifetime	Amount of time (seconds) that the connection can remain idle in the connection pool	60
Connection Reset	True: This particular connection will be put into the connection pool when it is closed. False: This particular connection will not be put into the connection pool when it is closed.	false
Enlist	True: Enlistment to Distributed Transaction Coordinator (DTC) is allowed. (This will enlist only if a COM+ transaction is in progress at connect time.) False: Enlistment to Distributed Transaction Coordinator (DTC) is not allowed.	true

Attribute	Description	Default value (if omitted)
Isolation Level; IsolationLevel	Isolation level for the connection. Possible values are: ReadCommitted ReadUncommitted RepeatableRead Serializable This keyword is supported only for applications participating in a distributed transaction, such as a COM+ application. For applications that do not participate in a distributed transaction, this keyword is not supported and an InvalidArgument exception is thrown.	
Max Pool Size	Maximum pool size	no maximum
Min Pool Size	Minimum pool size	0
Persist Security Info	True: Allow security-sensitive information, such as password, to be returned as part of the connection string after the connection has been opened or if the connection has ever been in an opened state. False: Security-sensitive information is not returned as part of the connection string. False is strongly recommended.	false
Pooling	Connection pooling switch (true/false)	true

Example 5-4 shows how to create a connection.

Example 5-4 Using DB2Connection

```
//Create connection string
string myConnString = "Server=myiSeries:446; Database=myRDB; UID=myUserid;
PWD=myPassword;";
//Create connection object
DB2Connection myConnection = new DB2Connection(myConnString);
//Connect to database
myConnection.Open();
```

Connections can be opened in two ways:

- Explicitly by calling the Open method on the connection
- Implicitly when using a DataAdapter

Table 5-5 and Table 5-6 on page 201 describe some important interfaces of the Connection object.

Table 5-5 Connection object properties

Public property	Description
ConnectionString	This is required for making a connection with a database. It requires the database source name and other parameters. For example, for DB2 .NET provider you can specify a ConnectionString property with Connection cnn as: cnn.ConnectionString ="Database=myDB2";

Table 5-6 Connection object methods

Public method	Description
Open	Opens a database connection that is specified in a <code>ConnectionString</code> property, such as: <code>cnn.Open();</code> The <code>Connection</code> object throws an exception if it fails to open a database connection.
Close	Used to close the database connection. For example: <code>cnn.Close();</code>
CreateCommand	Returns a <code>Command</code> object associated with the connection, which can be used to perform SQL operations on a database. For example: <code>DB2Command cmd = cnn.CreateCommand();</code>
BeginTransaction	Begins a transaction at the local level.

5.6.6 Using the DB2Command object

`DB2Command` represents a SQL statement or stored procedure to execute against a database. This command can use parameters and be executed in a transaction. Example 5-5 illustrates the use of the `DB2Command`.

Example 5-5 Using DB2Command

```
//Create connection string
string myConnString = "DATABASE=myDB2;";
//Create connection object
DB2Connection myConnection = new DB2Connection(myConnString);
//Connect to database
myConnection.Open();
//Create DB2Command using existing connection object
DB2Command cmd = myConnection.CreateCommand();
//Establish SQL statement to be executed
cmd.CommandText = "UPDATE SAMPLEDB.EMPLOYEE SET Salary = Salary * 1.05;";
//Create a transaction
DB2Transaction trans = cnn.BeginTransaction();
//Execute the command
cmd.ExecuteNonQuery();
//Complete the transaction
trans.Commit();
```

Table 5-7 and Table 5-8 on page 202 describe some important interfaces of the `Command` object.

Table 5-7 Command object properties

Public property	Description
CommandType	Describes whether the <code>Command</code> object will execute a SQL statement or Stored Procedure.
CommandText	Used to set or get a SQL statement or Stored Procedure to execute at a database. The default value of the <code>CommandType</code> property is <code>CommandType.Text</code> . For example: <code>cmd.CommandText = "select * from STAFF";</code> <code>cmd.CommandType = CommandType.Text;</code>

Table 5-8 Command object methods

Public method	Description
CreateParameter	Used for handling parameters. The parameter could be input-only, output-only, bidirectional, or a stored procedure return value parameter.
ExecuteNonQuery	Can be used to perform UPDATE, INSERT, or DELETE SQL operations on a database. This method returns the number of rows that are affected after executing the SQL statement. For example: <pre>cmd.Connection.Open(); rowsAffected = cmd.ExecuteNonQuery();</pre>
ExecuteReader	Used for reading results by executing a SELECT statement on a database.
ExecuteScalar	Used for retrieving a single value from a database. This reduces overhead required for the ExecuteReader method. For example: <pre>cmd.CommandText = "select count(*) from STAFF"; Int32 count = (int32) cmd.ExecuteScalar();</pre>

5.6.7 Using the DB2DataReader object

The DB2DataReader provides a way to read a forward-only stream of data rows from a database. Example 5-6 illustrates the use of the DB2DataReader object.

Example 5-6 Using DB2DataReader

```
//Create connection string
string myConnString = "DATABASE=myDB2;";
//Create connection object
DB2Connection myConnection = new DB2Connection(myConnString);
//Connect to database
myConnection.Open();
//Create DB2Command using existing connection object
DB2Command cmd = myConnection.CreateCommand();
//Establish SQL statement to be executed
cmd.CommandText
    = "SELECT FirstNme, LastName, HireDate FROM SAMPLEDB.EMPLOYEE ORDER BY HireDate ASC;";
//Create a DataReader
DB2DataReader reader;
//Execute the command to fill the DataReader
reader = cmd.ExecuteReader();
//Iterate the DataReader
while(reader.Read())
{
    //Retrieve data from reader row
    String myFirstName = reader.GetString(0);
    String myLastName = reader.GetString(1);
    DateTime myHireDate = reader.GetDate(2);
    //Make something with these values
    Console.WriteLine(myLastName.Trim()+"", "+myFirstName.Trim()
        +" - "+myHireDate.ToShortDateString());
}
//DataReader must be closed
reader.Close();
//Close the connection
myConnection.Close();
```

When we run this code snippet, we see the result shown in Figure 5-20 on page 203.

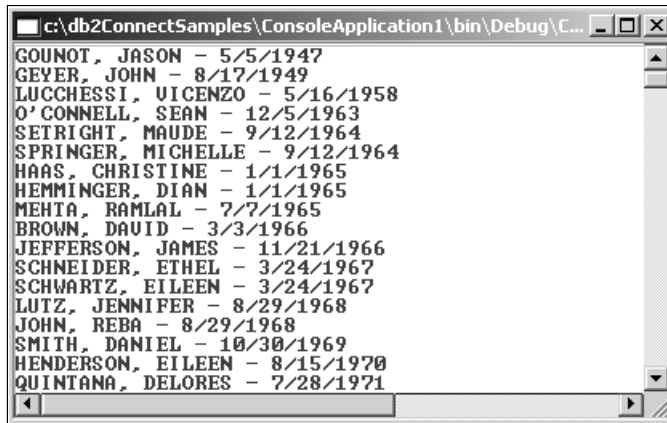


Figure 5-20 DB2DataReader sample output

Table 5-9 and Table 5-10 describe important properties and methods of the DataReader class.

Table 5-9 DataReader class properties

Public property	Description
FieldCount	Returns the number of columns in the current row.
HasRows	Indicates whether DataReader has one or more rows.

Table 5-10 DataReader methods

Public method	Description
Read	Used to read records one by one. This method automatically advances the cursor to the next record and returns true or false, indicating whether DataReader was able to read any rows.
Close	Closes the DataReader.
Getxxxx	Used to get data of type xxxx. For example, the GetBoolean method is used to get Boolean records, and the GetChar method is used to get char-type data.

5.6.8 Using the DB2DataAdapter object

The DB2DataAdapter represents a set of data commands and a connection to a database that are used to fill the DataSet and update the database. The DB2DataAdapter serves as a bridge between a DataSet and database for retrieving and saving data. The DB2DataAdapter provides this bridge by using Fill to load data from the database into the DataSet and using Update to send changes made in the DataSet back to the database.

We can use DB2DataAdapter to fill a DataSet as shown in Example 5-7. (Error control has been omitted for brevity.)

Example 5-7 Using DB2DataAdapter to retrieve a DataSet

```
private DataSet GetStaffDataSet()
{
    //Create connection string
    string myConnString = "DATABASE=myDB2;";
    //Create empty DataSet
    DataSet myDataset = new DataSet();
```

```

//Create connection object
DB2Connection myConnection = new DB2Connection(myConnString);
//Use DataAdapter to connect to database and retrieve results
DB2DataAdapter myAdapter = new DB2DataAdapter();
myAdapter.SelectCommand = new DB2Command("SELECT * FROM STAFF", myConnection);
myAdapter.Fill(myDataSet);
//Return filled DataSet
return myDataSet;
}

```

Table 5-11 and Table 5-12 show some important DataAdapter public properties and methods.

Table 5-11 DataAdapter properties

Public property	Description
DeleteCommand	Gets or sets a SQL statement or Stored Procedure to delete records from the data set. For example: <pre> DB2DataAdapter adpt = new DB2DataAdapter (); DB2Command cmd; cmd = new DB2Command("DELETE FROM Customers WHERE CustomerID = ' ', cnn); adpt.DeleteCommand = cmd; </pre>
InsertCommand	Inserts new records into a database using SQL or Stored Procedure.
SelectCommand	Selects records in a database using SQL or Stored Procedure.
UpdateCommand	Used to update records in a database using SQL or Stored Procedure.

Table 5-12 DataAdapter methods

Public method	Description
Fill	Used to fill records in DataSet. For example: <pre> adpt.Fill(dataset); //fills dataset </pre>
Update	Used to update rows in DataSet and a database by performing INSERT, DELETE, or UPDATE operations.

5.7 Advanced topics

In this section we cover advanced features of LUW .NET provider.

5.7.1 Using large objects (LOBs)

A LOB is a large block of data that is stored in a database, such as an image or sound file. Note that LOBs are not stored in table rows, but in separate pages referenced by a pointer in the row.

The three types of LOB data types in DB2 are:

- Character large objects (CLOBs)

These are typically used to store blocks of text items, such as ASCII or PostScript files.

- Double-byte character large objects (DBCLOBs)

A DBCLOB is used to store large DBCS character-based data such as documents written with a single character set. A DBCLOB value can be up to 1,073,741,823 double-byte characters long.

► Binary large objects (BLOBs)

BLOB has no structure that can be interpreted by the database management system but is known only by its size and location.

The IBM DB2 for LUW .NET provider has the capability to handle LOBs. By using this provider, code can be written to SELECT, INSERT, UPDATE, or DELETE LOBs from a database.

To demonstrate, we create an Employee Photo Viewer application. This application takes an employee name selected by the user and displays the employee's photo, which is stored as a BLOB data type in the EMP_PHOTO table.

The running Employee Photo Viewer application looks as shown in Figure 5-21.

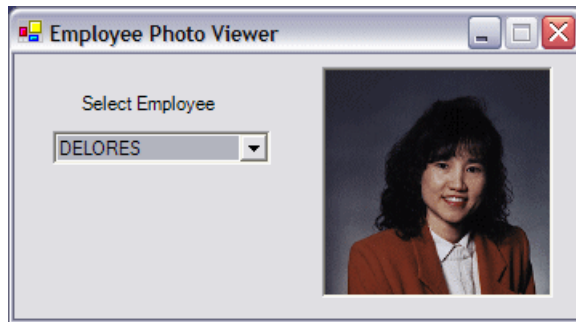


Figure 5-21 Accessing LOB data sample

To start, open a new Visual C# Windows application project and put controls on the default form. The code for this application is divided into two parts, the first for populating the combo box with user names, and the second for displaying the photo of an employee as per the selection in the combo box. Both parts use the connection object created by the getCnn() function. The code for the getCnn() function is shown in Example 5-8.

Example 5-8 Employee Photo Viewer getCnn method

```
private DB2Connection getCnn ()
{
    DB2Connection conn = null;
    try
    {
        conn=new DB2Connection("database=myDB2;Connect Timeout=30;user
Id=myuserid;password=mypassword;");
        conn.Open ();
    }
    catch (DB2Exception e)
    {
        MessageBox.Show (e.Message);
        conn.Close ();
    }
    return conn;
}
```

The application uses the combo box control to give the user a way to select an employee name. The ValueMember property is used in conjunction with the SelectedValue property to get the employee number that is associated with the employee's name.

The code snippet in Example 5-9 on page 206 shows how the combo box is populated for employee names, and the ValueMember property is used to assign an employee number.

Example 5-9 Employee Photo Viewer populateEmpNameCombo method

```
private void populateEmpNameCombo()
{
    DB2Connection cnn = null;
    cnn=getCnn();
    // Create the DataSet
    DataSet ds = new DataSet();
    // Fill the Dataset using Data Adapter
    DB2DataAdapter da1 = new DB2DataAdapter("SELECT distinct a.EMPNO,a.FIRSTNME FROM
    SAMPLEDB.EMPLOYEE a, SAMPLEDB.EMP_PHOTO b where a.empno = b.empno",cnn);
    DataTable dataTable = new DataTable();
    da1.Fill(dataTable);
    comboBox1.Items.Clear ();

    //Define the event handler for combo box SelectedValueChanged event
    this.comboBox1.SelectedValueChanged += new
    System.EventHandler(this.comboBox1_SelectedValueChanged);
    //assign combo box properties to populate the database.
    comboBox1.DataSource = dataTable;
    comboBox1.DisplayMember = "FIRSTNME";
    comboBox1.ValueMember = "EMPNO";
    cnn.Close();
}
```

The `populateEmpNameCombo()` method is invoked in the Form Load event, which populates the data when the application gets started, as shown in Example 5-10.

Example 5-10 Employee Photo Viewer populateEmpNameCombo method

```
private void Form1_Load(object sender, System.EventArgs e)
{
    populateEmpNameCombo();
}
```

When the user selects any name in the combo box, the `SelectedValueChanged` event is executed. In this event the LOB data for the employee photo is retrieved from the `EMP_PHOTO` table using the selected employee name. The `SelectedValue` property of the combo box returns the employee number associated with the selected employee name as shown in Example 5-11.

Example 5-11 Employee Photo Viewer comboBox1_SelectedValueChanged event handler

```
private void comboBox1_SelectedValueChanged(object sender, System.EventArgs e)
{
    if (comboBox1.SelectedIndex >= 0)
    {
        DB2Connection cnn = null;
        cnn=getCnn();
        DB2Command cmd = cnn.CreateCommand();
        cmd.CommandText =
            "SELECT picture FROM sampledb.emp_photo " +
            "where empno='" + comboBox1.SelectedValue + "'";
    }
}
```

The `ExecuteReader` method returns the result by executing the SQL statement against the `EMP_PHOTO` database as shown in Example 5-12 on page 207. The data reader is used to get the LOB data of the employee's photo.

Example 5-12 Employee Photo Viewer event handler continued

```
DB2DataReader reader;  
reader = cmd.ExecuteReader();//;
```

Next, the data reader is used with the MemoryStream object to consume the LOB data in stream format as shown in Example 5-13.

Example 5-13 Employee Photo Viewer event handler continued

```
if(reader.Read())  
{  
    int maxSize = 102400;  
    Byte[] out_picture = new Byte[maxSize];  
    reader.GetBytes(4,0,out_picture,0,maxSize);  
    MemoryStream mem = new MemoryStream(out_picture);
```

Finally, the employee image is displayed by assigning stream data from the MemoryStream object using the FromStream method of the Image object, as shown in Example 5-14.

Example 5-14 Employee Photo Viewer event handler continued

```
        picEmp.Image =Image.FromStream(mem);  
    }  
    reader.Close();  
    cnn.Close();  
}  
}
```

5.7.2 Using the DB2CommandBuilder object

The DB2CommandBuilder object provides dynamic updating logic for the data in a dataset object. Methods of the DB2CommandBuilder object take care of transact SQL statements (INSERT, UPDATE, and DELETE) without writing any SQL statements. To achieve this, DB2CommandBuilder methods use schema information of a table specified using SelectCommand. Using this table schema information, it automatically builds INSERT, UPDATE, and DELETE command objects and required SQL statements. The command builder object is used in association with data adapter object. When data adapter invokes update method, the automatically built command object gets invoked.

Basic requirements for using the DB2CommandBuilder

The DB2CommandBuilder can generate updating logic if all of the following are true:

- ▶ The query returns data from only one table.
- ▶ A table should have a primary key.
- ▶ The primary key is included in the results of the query.
- ▶ The primary key ensures that the query-based updates that the DB2CommandBuilder generates can update one row at most.
- ▶ The SelectCommand must be specified. The DB2CommandBuilder object uses the DB2DataAdapter object's SelectCommand property to fetch the metadata necessary for the updating logic.

In this example of DB2CommandBuilder and how it works, we show how to update the SALARY of an employee without writing an update SQL statement.

1. Create a new Console Application in C# and import the necessary namespaces (Example 5-15).

Example 5-15 C# namespaces

```
using IBM.Data.DB2;  
using System.Data;
```

2. Declare connection, data adapter, and command builder objects and open a connection to the database as shown in Example 5-16. Note that the command builder object is created by passing a data adapter object. The command builder object automatically generates transact SQL statements (INSERT, UPDATE, and DELETE), depending on the Select SQL statement passed to the data adapter object.

Example 5-16 DB connection code

```
DB2Connection cnn = new DB2Connection("server=19.15.192.121:446;database=myDB2;Connect  
Time-out=30;user Id=myUID;password=myPWD;");  
//select SQL statement is also get used by command builder to build schema information  
DB2DataAdapter dADP = new DB2DataAdapter("select * from sampled.b.employee where  
empno='000010'", cnn);  
DB2CommandBuilder cBLD = new DB2CommandBuilder(dADP);  
cnn.Open();
```

3. Declare a dataset object and fill the data in it using data adapter (Example 5-17).

Example 5-17 Dataset object declaration

```
//declare dataset object and fill with employee information  
DataSet empDS = new DataSet();  
dADP.Fill(empDS, "EMPLOYEE");
```

4. For verification purposes, print the existing salary of the employee, then set the salary to the new value as shown in Example 5-18.

Example 5-18 Data shown on console

```
//salary before update  
Console.WriteLine ("Employee Salary before update " +  
empDS.Tables["EMPLOYEE"].Rows[0]["SALARY"]);  
//new salary  
empDS.Tables["EMPLOYEE"].Rows[0]["SALARY"]=92345.39;
```

5. Salary in a data set row can be updated using the Update() method of the data adapter (see Example 5-19). The Update() statement calls the respective INSERT, UPDATE, or DELETE statements for each inserted, updated, or deleted row in the DataSet.

Example 5-19 Update method

```
//update salary  
dADP.Update(empDS, "EMPLOYEE");
```

6. Verify the updated salary and close the connection (Example 5-20).

Example 5-20 Update verification and close connection

```
//check updated salary
Console.WriteLine ("Employee Salary after update " +
empDS.Tables["EMPLOYEE"].Rows[0]["SALARY"]);
//close connection
cnn.Close();
```

Advantages and disadvantages of using the CommandBuilder

The CommandBuilder object has advantages and disadvantages, which we discuss here.

Advantages

- ▶ The CommandBuilder object requires less code for updating transact SQL statements. It enables you to generate updating logic without writing any SQL statements for UPDATE, INSERT, and DELETE queries.
- ▶ The CommandBuilder is also useful in any application where you need to support updating, but you will not know the structure of your queries at design time.
- ▶ You can also generate transact SQL statements using the CommandBuilder object. In the previous example, you can get an Update SQL statement using following statements:

```
string UpdateSQLStatement = cBLD.GetUpdateCommand().CommandText
Console.WriteLine (UpdateSQLStatement);
```

- ▶ From that SQL statement, you can also check and construct values of various properties on the Parameter objects it constructs.

Disadvantages

- ▶ For using the CommandBuilder object, you should follow the basic requirements listed in “Basic requirements for using the DB2CommandBuilder” on page 207.
- ▶ A CommandBuilder will not help you submit updates using stored procedures.
- ▶ Due to dynamic generation of SQL statements, the CommandBuilder does not offer the best possible run-time performance.

5.7.3 Performing transactions

DB2 LUW provider supports transactions in conjunction with the .NET code. A transaction is a set of related tasks that either succeeds (commits) or fails (aborts) as a unit. In .NET you can perform transactions in two ways, depending on your environment:

- ▶ Local transaction
- ▶ Distributed transaction

Figure 5-22 on page 210 illustrates the different ways to perform transactions in .NET.

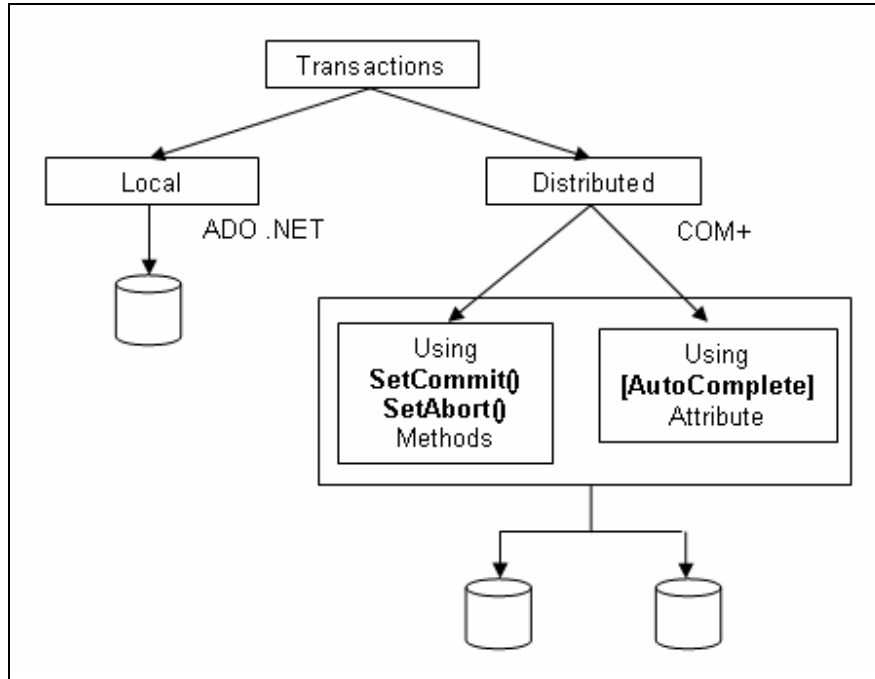


Figure 5-22 Different ways to perform transactions in .NET

Local transactions

Local transactions are performed on a single database and can be performed by using ADO .NET. In the local transaction model, the code uses the transaction support features of either ADO.NET or Transact-SQL directly in component code or stored procedures, respectively.

Local transaction enables you to explicitly begin a transaction, control each connection and resource enlistment within the transaction boundary, determine the outcome of the transaction (commit or abort), and end the transaction.

Distributed transactions

Distributed transactions are performed in a distributed environment that includes one or more databases, such as multiple resource managers located on remote computers.

Distributed transactions can be performed by registering the .NET assembly with COM+. The .NET Framework provides various declarative attributes that specify how to participate in the transaction. This model enables you to easily configure multiple components to perform work within the same transaction.

Distributed transactions can be performed by using the `SetAbort()` and `SetCommit()` methods, or by using the `AutoComplete` attribute.

The IBM DB2 for LUW .NET provider can handle both local and distributed transactions, which we discuss in the next section.

Loosely-coupled transactions are not supported

The Microsoft COM+ transactional environment expects that a resource (such as DB2) that has two concurrent branches of the same global transaction would treat these two branches as a common transaction. In the case of DB2, that implies that locks would be shared between these two branches as best as possible. The reason for this required behavior is that COM+ can either use different connection to execute different portions of a common

transaction. Because different connections are used, Microsoft XA Transaction Manager uses differing branch qualifiers for each connection. Additionally, it is quite possible that COM+ objects executed within the same global transaction are dispersed across different physical machines, so different connections are required. Because the DB2 UDB for iSeries does not support loosely coupled distributed transactions as of this writing, this support is not available through the DB2 for LUW .NET provider when using an iSeries.

Performing local transactions with the DB2 LUW provider

IBM DB2 for LUW .NET provider has the capability to handle transactions at the local level. You can define where to start a transaction using the BeginTransaction method of the connection object, and you can control where to commit and roll back results using the Commit and RollBack methods respectively.

The DB2Transaction class is used for controlling local transactions. In the following example, the UPDATE SQL query updates the STAFF table, a transaction object commits the update using the Commit method, and the Rollback method rolls back the entire transaction if any errors occur.

To understand how the local transaction works, try this example:

1. Create a console application in Visual Studio .NET, add a reference to the DB2 provider, and import the necessary namespaces. In a code window declare the data objects as shown in Example 5-21.

Example 5-21 Performing local transactions with DB2 LUW provider: declaring objects

```
DB2Connection cnn = null;
DB2Transaction trans = null;
DB2Command cmd = null;
```

2. The connection and command object are created with CommandText for updating the record in the table, as shown in Example 5-22.

Example 5-22 Performing manual transactions with DB2 LUW provider: creating objects

```
try
{
    cnn = new DB2Connection(("database=myDB2;user Id=myuserid;password=mypassword;"));
    cmd = new DB2Command();
    cmd.Connection = cnn;
    cmd.CommandText = "update sampled.b.staff set salary = 20000 where id =20";
    cnn.Open();
```

3. In the next statement, the BeginTransaction method of the connection object is used to initiate the transaction object, then it is assigned to the Transaction property of the command object as shown in Example 5-23.

Example 5-23 Performing manual transactions with DB2 LUW provider: initiating a transaction

```
trans = cnn.BeginTransaction ();
cmd.Transaction = trans;
```

4. The ExecuteNonQuery method executes update statements on the database and returns the total number of updated rows as shown in Example 5-24.

Example 5-24 Performing manual transactions with DB2 LUW provider: executing the statement

```
int rowsUpdated = cmd.ExecuteNonQuery();
Console.WriteLine ("Rows Updated = " + rowsUpdated);
```

If everything runs as planned, the Commit method of the transaction object commits the update in the STAFF table, as shown in Example 5-25.

Example 5-25 Performing manual transactions with DB2 LUW provider: committing the transaction

```
trans.Commit ();  
cnn.Close ();  
}
```

However, if any errors occur, the transaction is rolled back using the Rollback method shown in Example 5-26.

Example 5-26 Performing manual transactions with DB2 LUW provider: committing the transaction

```
catch (Exception e)  
{  
    trans.Rollback();  
    Console.Out.WriteLine(e.Message);  
}
```

Performing distributed transactions with the DB2 LUW provider

Distributed transaction in .NET can be performed using COM+, which provides an enterprise development environment, based on COM technology, for creating component-based, distributed applications. In .NET, the System.EnterpriseServices namespace provides the necessary infrastructure for enterprise applications. With the help of this namespace a .NET object can access COM+ services.

Covering the entire System.EnterpriseServices namespace is beyond the scope of this book, but to understand how COM+ works and some basic concepts, we now go through the sample code for creating a serviced component.

Creating a COM+ component

To use a .NET assembly as COM+ component, you should follow these steps:

- ▶ Apply the TransactionAttribute to the class.

The TransactionAttribute attribute specifies the type of transaction or a transaction support level that is available to the attributed class. This attribute accepts as a parameter a value from an enumeration in the System.EnterpriseServices namespace called TransactionOption and supports any of the following automatic transaction values:

- Disabled

The object should ignore any transaction in the current context.

- NotSupported

Specifies that the object should create the component in a context with no governing transaction.

- Required

Specifies that the object should share a transaction if one exists, or create a new transaction if necessary.

- RequiresNew

Specifies that the object should create the component with a new transaction, regardless of the state of the current context.

- Supported

Indicates that the object should share a transaction if one exists.

The Transaction attribute must be applied at the class level, as in the code snippet shown in Example 5-27.

Example 5-27 Applying the transaction attribute

```
[Transaction(TransactionOption.Supported)]
public class demoClass
{
}
```

- ▶ Derive your class from the ServicedComponent Class.
- ▶ Sign the assembly with a strong name.
- ▶ Register the assembly that contains your class with the COM+ catalog.

In .NET, the COM+ transaction can be controlled using two approaches:

- ▶ Using SetComplete and SetAbort methods

With this approach, you can use either the SetComplete or SetAbort method of the System.EnterpriseServices.ContextUtil class to explicitly commit or abort a transaction, as shown in Example 5-28.

Example 5-28 Using SetComplete and SetAbort methods

```
if( !TransactionSucess() )
{
    //Something goes wrong.
    ContextUtil.SetAbort();
}
else
{
    //All goes well.
    ContextUtil.SetComplete();
}
```

- ▶ Using AutoComplete

You can control the transactional behavior of the COM+ class methods by using the AutoComplete attribute. In this approach, the method marked with the AutoComplete attribute explicitly calls SetComplete() if the method works without any exception; otherwise, it makes an explicit call to SetAbort(). The code snippet in Example 5-29 shows how to write an AutoComplete attribute for a method.

Example 5-29 Using SetComplete and SetAbort methods

```
public class demoClass
{
    [AutoComplete]
    public demoMethod()
    {
    }
}
```

Using the SetAbort and SetComplete methods to control transactions

The following steps explain how to develop, configure, and install COM+ components using the SetAbort and SetComplete methods. Note that the steps for creating a COM+ component using the SetAbort and SetComplete methods or AutoComplete attribute are the same except for the code.

1. COM+ services are bundled into an assembly (.dll) or into a class library. Therefore, to create a COM+ component, you have to create a new Class Library project in Visual Studio .NET. Start Visual Studio .NET and create a new Class Library project with the name db2TransServer in Visual C# Projects types as shown in Figure 5-23.

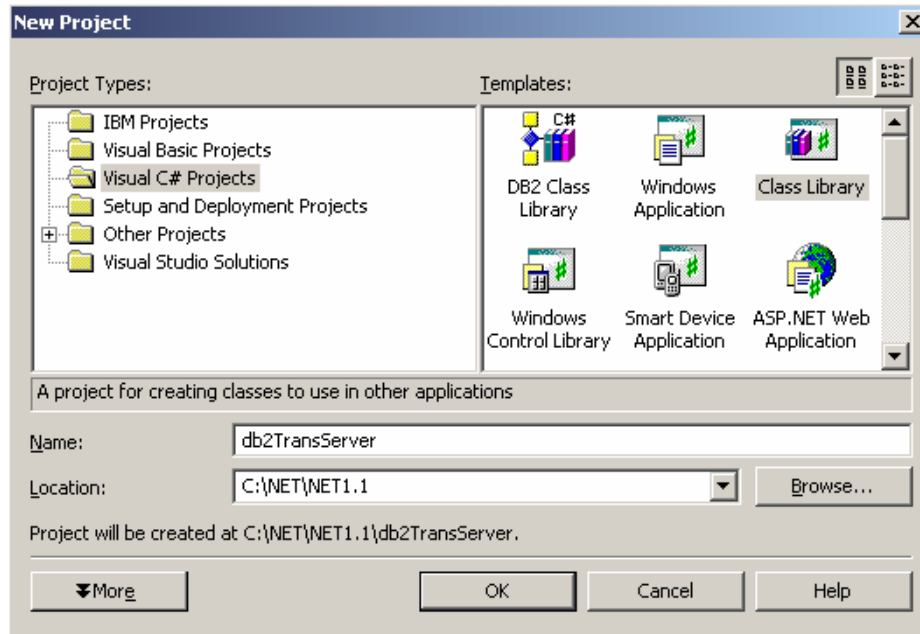


Figure 5-23 File → New → Project (creating a new class library project)

2. Visual Studio .NET creates a new class library project with default files. Rename Class1.cs to updateEmployee.cs in the Solution Explorer window, and change the class name and default constructor from Class1 to updateEmployee.
3. From the Visual Studio .NET menu bar, select **Project** → **Add References** and add references to the IBM.Data.DB2 and System.EnterpriseServices components (Figure 5-24 on page 215).

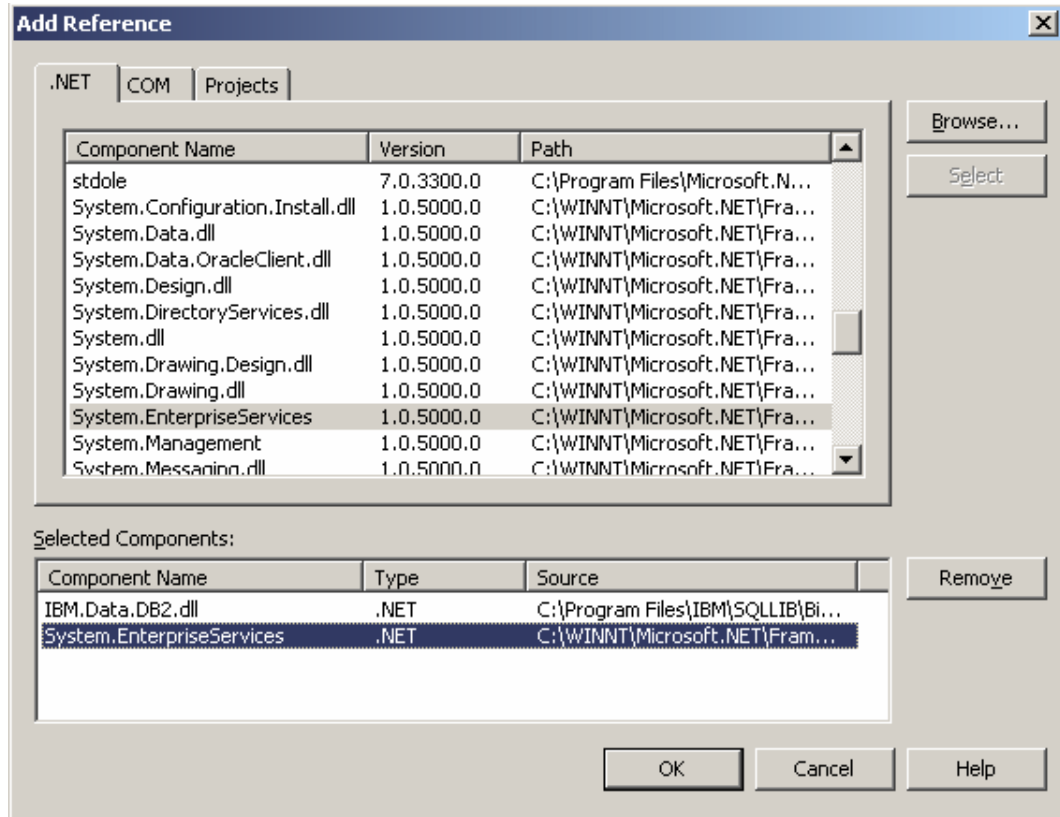


Figure 5-24 Adding references

4. In the code window of the `updateEmployee` class, include the necessary namespaces as shown in Example 5-30.

Example 5-30 Including namespaces

```
using System;
using System.Data ;
using System.EnterpriseServices ;
using IBM.Data.DB2;
using System.Runtime.CompilerServices;
using System.Reflection;
```

The `System.EnterpriseServices` namespace handles the transaction operations. The `System.Runtime.CompilerServices` and `System.Reflection` namespaces are used for declaring assembly-related information.

5. To use an assembly as a COM+ component, give it a strong name using the `ApplicationName` and `AssemblyKeyFileAttribute` attributes as shown in Example 5-31. A strong name consists of the assembly's identity—its simple text name, version number, and culture information (if provided)—plus a public key and a digital signature. Note that you can write this information in the `AssemblyInfo.cs` file. Be sure the code does not have entries in both code and `AssemblyInfo.cs` files, or a compiler error will occur.

Example 5-31 Using `ApplicationName` and `AssemblyKeyFileAttribute` attributes

```
// Supply the COM+ application name.
[assembly: ApplicationName("db2TransServer")]
// Supply a strongly-named assembly.
[assembly: AssemblyKeyFileAttribute(@"a\db2TransServerCrypt.snk")]
```

6. The Strong Name tool (sn.exe) helps to sign assemblies with strong names by generating a public/private key pair. This public and private cryptographic key pair is used during compilation to create a strongly named assembly. sn.exe provides options for key management, signature generation, and signature verification. The key pair files usually have a .snk extension. You can generate a key pair from a command prompt by specifying the file location and name, as shown in Example 5-32.

Example 5-32 Using the sn tool to generate a key pair file

```
c:\WINNT\Microsoft.NET\Framework\v1.1.4322>sn -k c:\a\db2TransServerCrypt.snk
```

```
Microsoft (R) .NET Framework Strong Name Utility Version 1.1.4322.573  
Copyright (C) Microsoft Corporation 1998-2002. All rights reserved.
```

```
Key pair written to c:\a\db2TransServerCrypt.snk
```

```
C:\WINNT\Microsoft.NET\Framework\v1.1.4322>
```

Be sure to specify the path information of the .snk file when specifying the AssemblyKey-FileAttribute attribute. In our example, we store the .snk file at c:\a, which is reflected in the attribute parameter as (@"\a\db2TransServerCrypt.snk")

7. Specify additional attributes. The System.EnterpriseServices namespace has several COM+ related .NET attributes that control the behavior of the class when participating in transactions.

For example, in our code we use the ApplicationAccessControl attribute to indicate whether security can be configured for an assembly, as shown in Figure 5-33. Because we are not applying any security, we set the value to false. (If you do not specify this property, the security configuration is enabled by default.)

Example 5-33 Disabling COM+ security configuration

```
//Disable COM+ security configuration  
[assembly: ApplicationAccessControl(false)]
```

Similarly, the ApplicationActivation attribute is used to specify whether the class should be added to a library or a server COM+ application. The attribute takes a parameter of a type of enumeration called ActivationOption. The ActivationOption enumeration supports the following values:

- Library, which specifies a COM+ library application
- Server, which specifies a COM+ server application

The default value is Library but in our example we use Server activation by specifying the attribute as shown in Example 5-34.

Example 5-34 Setting Application activation

```
//Set Application activation to activated object in a system-provided process  
[assembly: ApplicationActivation(ActivationOption.Server)]
```

8. Insert the Transaction attribute [Transaction(TransactionOption.Required)] before the class declaration to indicate that the transaction is required for the class (Figure 5-35).

Example 5-35 Inserting the Transaction attribute

```
namespace db2TransServer
{
    [Transaction(TransactionOption.Required)]
    public class updateEmployee : ServicedComponent
    {
        public updateEmployee()
        {
        }
    }
}
```

9. This example gives functionality to update the employee salary by specifying the employee number and new salary. To fulfill this requirement, the updateEmpSalary method is declared, as shown in Example 5-36.

Example 5-36 updateEmpSalary method

```
public bool updateEmpSalary(string empNo, decimal salary)
{
}
}
```

10. Declare the DB2 connection and command objects for performing database operations. The UPDATE query is specified while declaring the command object that updates the salary for a given employee number, as shown in Figure 5-37.

Example 5-37 updateEmpSalary method continued

```
DB2Connection cnn = new DB2Connection ("database=myDB2;user
Id=myuserid;password=mypassword;");
DB2Command cmd = new DB2Command ("update sampledb.employee set salary=" + salary +
    "where empno='" + empNo + "'",cnn);
```

11. The next part of the code controls the transaction using the SetComplete and SetAbort methods. Here the ExecuteNonQuery method executes the SQL statement specified by the command object and returns the number of rows updated. In the following try-catch block, the logic is designed in such a way that the result gets committed only if there is one row updated. If any error occurs then the transaction is simply aborted. The code is shown in Figure 5-38.

Example 5-38 updateEmpSalary method continued

```
try
{
    cnn.Open();
    int rowsUpdated = cmd.ExecuteNonQuery();
    if (rowsUpdated == 1)
    {
        ContextUtil.SetComplete();
        return true;
    }
    else
    {
        //UPDATE failed
        ContextUtil.SetAbort();
        throw new Exception("Invalid account.");
    }
}
catch (Exception exc)
```

```

    {
        ContextUtil.SetAbort();
        throw new Exception(exc.Message + " Set Abort Called..");
    }
    finally
    {
        cmd.Dispose();
        cnn.Close();
    }
}

```

12. When the coding is complete, compile the code and build the assembly (.dll file). In order to use this .dll file with COM+ you have to register it and load it into the COM+ application. This can be achieved using a .NET Services Installation tool (Regsvcs.exe). This tool performs the following actions:

- a. Loads and registers an assembly.
- b. Generates, registers, and installs a type library into a specified COM+ application.
- c. Configures services coded in the class.

Example 5-39 shows how the Regsvcs utility is used to configure the .NET assembly as a COM+ component.

Example 5-39 Using .NET Services Installation tool

```

c:\WINNT\Microsoft.NET\Framework\v1.1.4322>regsvcs
C:\NET\NET1.1\db2TransServer\bin\Debug\db2TransServer.dll
Microsoft (R) .NET Framework Services Installation Utility Version 1.1.4322.573
Copyright (C) Microsoft Corporation 1998-2002. All rights reserved.

```

Installed Assembly:

```

Assembly: C:\NET\NET1.1\db2TransServer\bin\Debug\db2TransServer.dll
Application: db2TransServer
TypeLib: C:\NET\NET1.1\db2TransServer\bin\Debug\db2TransServer.tlb

```

```

C:\WINNT\Microsoft.NET\Framework\v1.1.4322>

```

After registering and configuring the assembly, open the COM+ MMC (Microsoft Management Console) (Figure 5-25 on page 219) by selecting **Start** → **Settings** → **Control Panel** → **Administrative Tools** → **Component Services**.

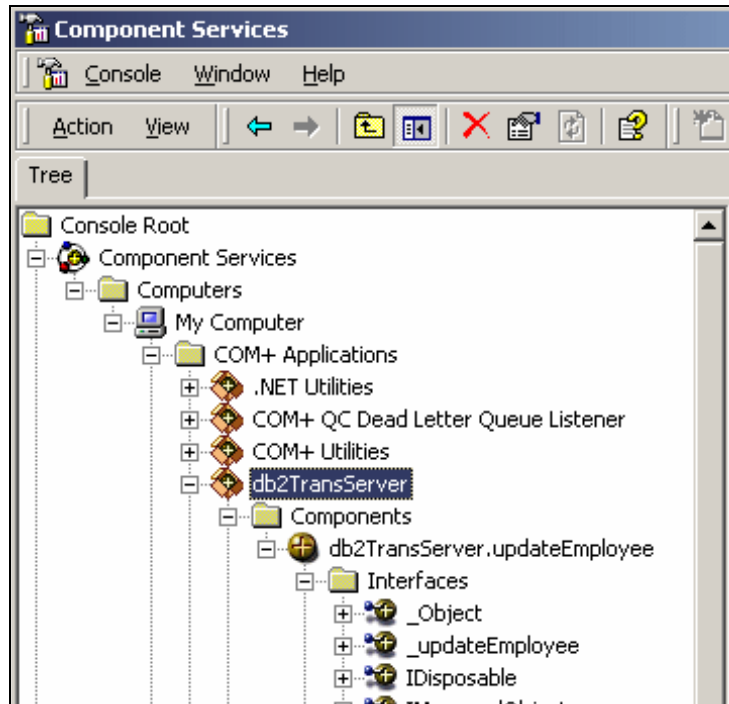


Figure 5-25 Component Services window

Using the AutoComplete attribute to control transactions

In addition to using the SetAbort and SetComplete methods to control transactions, you can use the AutoComplete attribute to control the transaction. The same code for updating the employee salary that we discussed in the previous section can be written using the AutoComplete attribute, as shown in Example 5-40.

Example 5-40 updateEmpSalary method continued

```
[AutoComplete]
public bool updateEmpSalary(string empNo, decimal salary)
{
    DB2Connection cnn =
        new DB2Connection ("database=myDB2;Connect Timeout=30;user
Id=myuserid;password=mypassword;");
    DB2Command cmd = new DB2Command ("update sampledb.employee set salary=" +
        salary + " where empno='" + empNo + "'",cnn);

    try
    {
        cnn.Open();
        cmd.ExecuteNonQuery();
    }
    catch (Exception exc)
    {
        throw new Exception(exc.Message + " Set Abort Called..");
    }
    finally
    {
        cmd.Dispose();
        cnn.Close();
    }
}
```

Testing the COM+ assembly

After building the serviced component, you can create a client application for testing. To demonstrate, we create a simple console application to access the COM+ component.

The steps for creating a client application involve:

1. Export and install proxy (only on remote machine).
2. Write client code by adding references to proxy and EnterpriseServices namespace.

Export and Install proxy

This step involves two tasks, one to export the proxy from the server and one to install the proxy on the client.

Note: This step is required only if you want to access the serviced component from a remote machine. If you are testing a client application on the same machine (such as in a development environment) where the serviced component is developed and registered, this step is not required.

To export the proxy from the server machine, open Component Services by selecting **Start → Settings → Control Panel → Administrative Tools → Component Services**.

1. Create an export package by right-clicking the component (**db2TransServer**) in the Component Services tree and selecting **Export**, as shown in Figure 5-26.

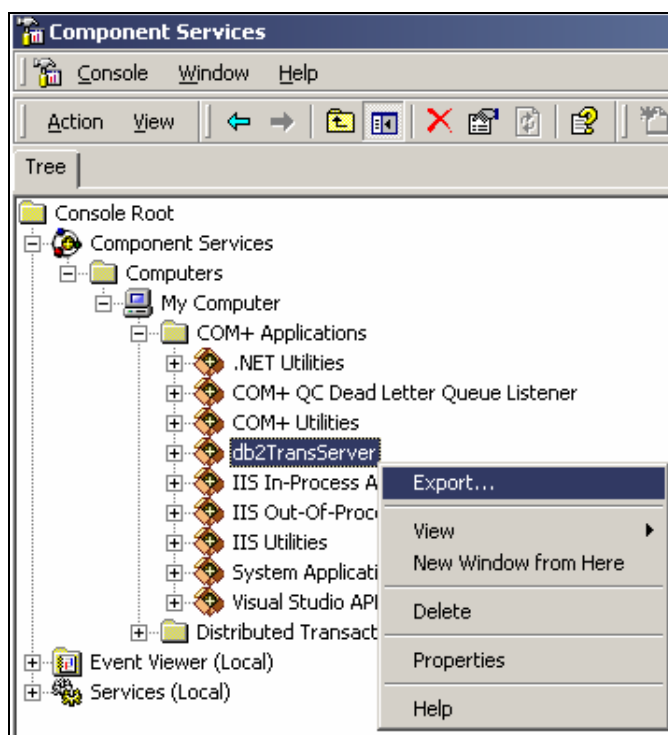


Figure 5-26 Exporting COM+ application

2. This opens the export wizard. Click **Next**.

3. In the next wizard screen, click **Browse** and specify the file name and location for exporting the COM+ component, as shown in Figure 5-27.

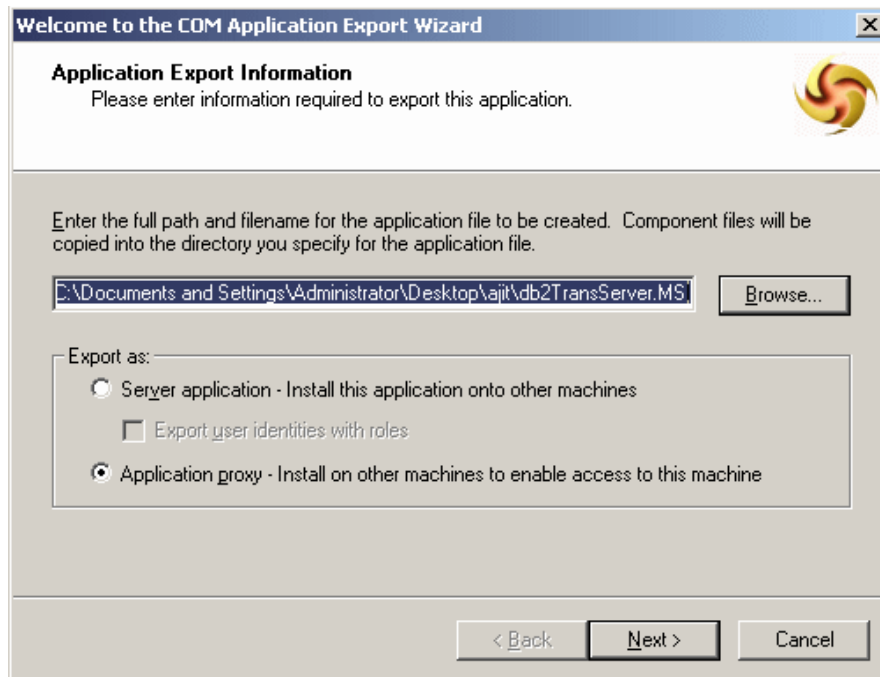


Figure 5-27 Wizard for Exporting COM+ application

4. Click **Next** and the window in Figure 5-28 opens. Click **Finish**.

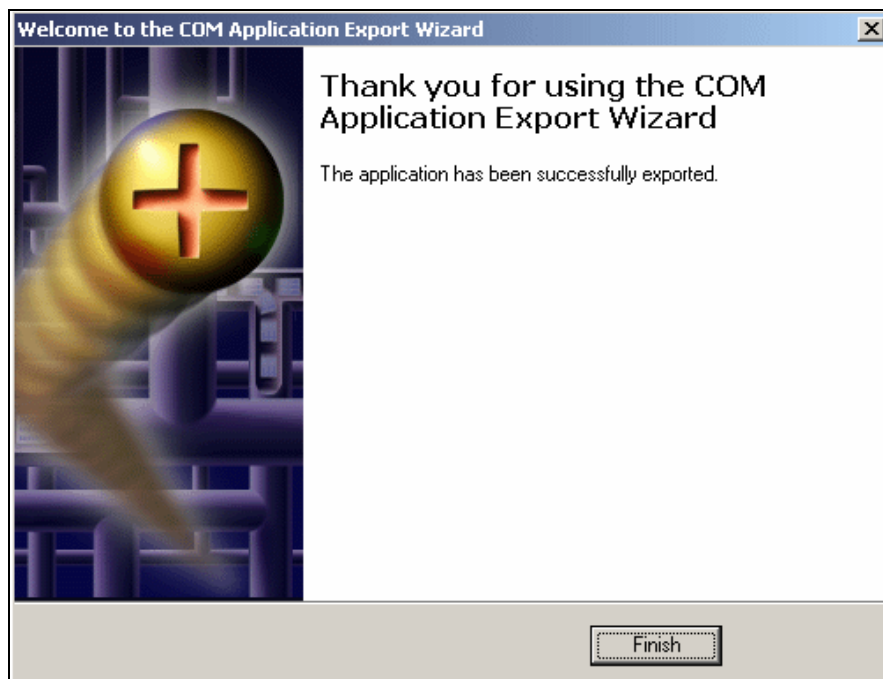


Figure 5-28 Successful Exporting of a COM+ application

These steps create an .MSI file (db2TransServer.MSI) and a .CAB file (db2TransServer.MSI.cab) in the specified path.

To install the proxy on a client, copy and paste the .MSI file and CAB files onto the client machine and run the .MSI file (db2TransServer.MSI). You can verify the proxy dll in the C:\Program Files\ComPlus Applications directory with a unique key folder as shown:

C:\Program Files\ComPlus Applications\{603D5F4E-A5C1-4D6D-A6B3-8B34898173EC}

Writing client code

To write client code, create a console application and add a reference to the System.EnterpriseServices namespace. This namespace is required for accessing serviced (COM+) components. To access a serviced component you also have to add a reference to the proxy dll. As we discuss in the previous section, when you install an exported package (.MSI file) on a client machine, a proxy dll is created in C:\Program Files\ComPlus Applications. Add a reference to the proxy by clicking **Browse** and selecting the component from the C:\Program Files\ComPlus Applications directory as shown in Figure 5-29.

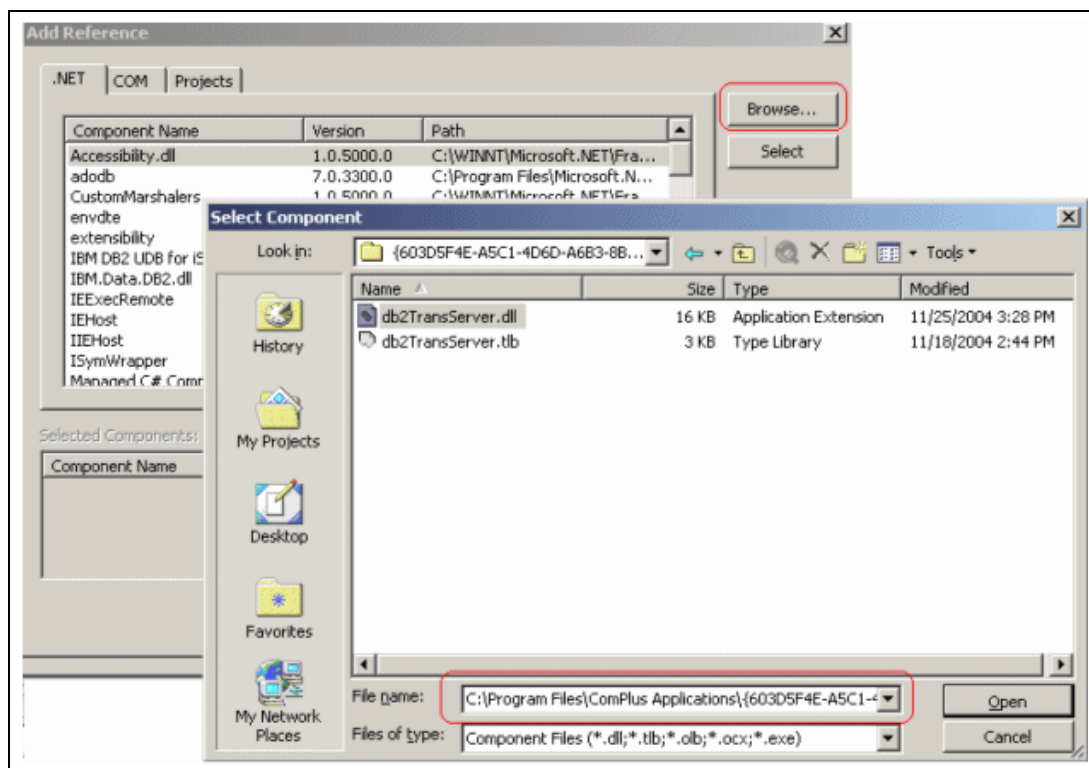


Figure 5-29 Importing a reference for the COM+ proxy

To access the COM+ method we need only a few lines of code to create an instance of the COM+ component and to access the updateEmployee method, as shown in Example 5-41.

Example 5-41 Accessing COM+ method

```
db2TransServer.updateEmployee ts = new db2TransServer.updateEmployee();
ts.updateEmpSalary("000010",99999.0);
```

5.8 Best practices

In this section we discuss some things you can do to improve the performance of your DB2 connections.

5.8.1 Connection pooling

Establishing and severing connections to the database server can be a resource-intensive process that adversely affects both PC and DB2 server performance.

To reduce this overhead, DB2 uses connection pooling to maintain open connections to the database in a readily accessible pool. Connection pooling is a technique that enables reuse of an established connection infrastructure for subsequent connections. Connection pooling helps to reduce the overhead of database connections and handles connection volume. This increases the scalability of applications by optimizing the use of host database servers.

How connection pooling works

When an application attempts to Open a database connection, the resource manager (or agent) checks for available connections in the pool. If a connection is not available in the pool, the manager creates a new connection for the requesting application. When the application issues a Close request, the manager does not pass this request along to the DB2 server. Instead, the manager returns the connection to the pool. The manager owns its connection to the DB2 server and the corresponding DB2 thread. When another application issues an Open request, the available connection thread in the pool is assigned to this new application. To ensure a secure operation, user identity information is passed along to the DB2 thread, which in turn performs user authentication.

Connection pooling is very helpful in Web and client/server environments where the frequency of connections and disconnections is high.

.NET and DB2 connection pooling

When you use the DB2 .NET provider, connection pooling is on by default. However, you can control the pooling mechanism by setting parameters in the `ConnectionString` property of `Db2Connection`. Table 5-13 shows some of the key parameters of the `ConnectionString` property that relate specifically to pooling.

Table 5-13 ConnectionString parameters specific to connection pooling

Parameter	Default value	Description
Connection Lifetime	60	Amount of time (in seconds) that the connection can remain idle in the connection pool
Connection Reset	false	If true, this particular connection will be put into the connection pool when it is closed. If false, this particular connection will not be put into the connection pool when it is closed.
Max Pool Size	no maximum	Indicates maximum pool size
Min Pool Size	0	Indicates minimum pool size
Pooling	true	Connection pooling switch (true/false)

The DB2Connection also has a ReleaseObjectPool method, which indicates that the resources of the connection pool can be released when the last underlying connection is released. This method is useful when you want the connection object to not be used again. When all connections in the pool are closed, the pool can be disposed of. Note that calling the method alone does not actually release the active connections that exist in the pool.

Before the pool is finally disposed, this must occur:

1. Call Close to release the DB2Connection object from the environment.
2. Allow each connection object to time out.
3. Call ReleaseObjectPool.
4. Invoke garbage collection.

Conversely, if you call Close on all active connections, and invoke garbage collection but do not call ReleaseObjectPool, the resources reserved for the pool will remain available.

After a pool is released, a request for a new DB2Connection creates a new pool.



Selecting the .NET provider

Choosing the appropriate .NET data provider for your application depends on your environment. In this chapter we provide a list of recommendations for selecting the right provider depending on your application's architecture and its functional requirements.

Currently, any of four providers can be used to access DB2 UDB for iSeries from .NET applications:

- ▶ **DB2 UDB for iSeries .NET provider (IBM.Data.DB2.iSeries)**
The ADO.NET-managed provider that is included with iSeries Access for Windows starting with V5R3M0.
- ▶ **DB2 UDB .NET provider (IBM.Data.DB2)**
The DB2 for Linux, UNIX, and Windows (LUW)–managed provider implemented by IBM software group.
- ▶ **Microsoft.Data.Odbc**
The Microsoft-supplied ODBC bridge provider that uses the iSeries Access for Windows ODBC driver for underlying database connectivity.
- ▶ **System.Data.OleDb**
The Microsoft-supplied OLEDB bridge provider using an iSeries Access for Windows OLE DB provider (IBMDA400, IBMDASQL, IBMDARLA) for underlying database connectivity.

Note: The OleDb bridge to IBMDA400 or IBMDARLA has not been tested for use with the Data Queue, Remote Command and Program Call, and Record Level Access support. This chapter assumes that the OleDb bridge is used to perform only SQL operations.

The .NET Framework includes the OLE DB .NET Data Provider and the ODBC .NET Data Provider. To use these providers, you must install the IBM iSeries Access for Windows product and select the appropriate Data Access component (ODBC or OLE DB Provider component). IBM DB2 managed providers are discussed at length in Chapter 4, “IBM DB2 UDB for iSeries .NET provider” on page 33 and in Chapter 5, “IBM DB2 for LUW .NET provider” on page 177. Here we briefly cover the functionality of the two bridge providers shipped with the .NET Framework.

6.1 ODBC .NET Data Provider

The ODBC .NET provider is a bridge that handles calls from .NET into a traditional ODBC driver. Specifically, the ODBC .NET Data Provider can be used with the iSeries Access for Windows ODBC driver.

The ODBC bridge involves jumping in and out of the .NET Framework environment for every interface call because, from the .NET point of view, the ODBC driver constitutes *unmanaged code*, which means that it has been compiled directly into a binary executable. *Managed code* is compiled into a .NET assembly that can be executed in the context of .NET Common Language Runtime (CLR). For managed code to call unmanaged code, marshalling of data must take place, and this can affect overall performance.

Similar to the IBM DB2 UDB for iSeries .NET provider, the iSeries Access ODBC driver uses the highly optimized, proprietary protocol to connect to the back-end database server job on the iSeries. The ODBC client sends SQL requests over the network to an iSeries server job. The server job runs the SQL requests on behalf of the client, and the results are reformatted and marshaled to the client. The iSeries database server jobs are called QZDASOINIT, and typically run in the QUSRWRK subsystem. This is illustrated in Figure 6-1.

Note: When using a secure connection, the iSeries database server job is called QZDASSINIT.

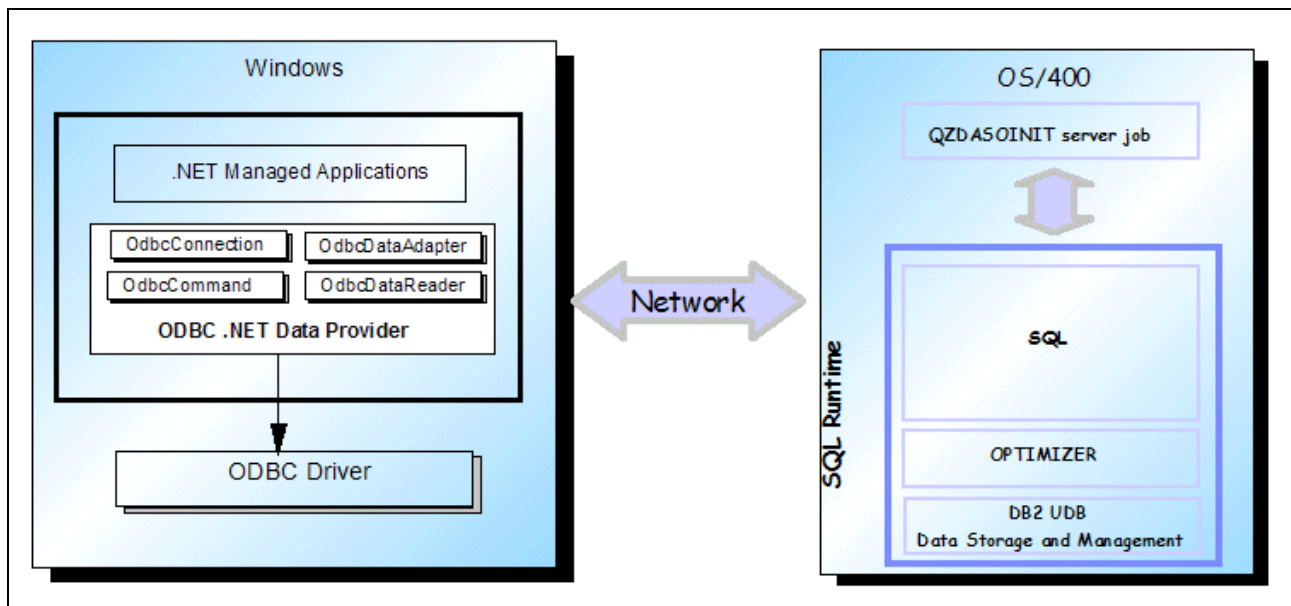


Figure 6-1 ODBC .NET provider accessing the DB2 UDB for iSeries database

The iSeries ODBC driver supports connection keywords that you can use to control the connection properties and to improve the application's performance. The complete list of the connection keywords can be found at the iSeries Information Center Web site at:

<http://publib.boulder.ibm.com/pubs/html/as400/infocenter.html>

As mentioned in Chapter 4, "IBM DB2 UDB for iSeries .NET provider" on page 33, several database-related functions are not supported at this time by the DB2 UDB for iSeries .NET

provider. However, most of this functionality is available through the iSeries ODBC driver. As a reminder, here is the list of the additional features that are supported by the ODBC driver:

- ▶ Datalink data type
- ▶ Extended dynamic SQL (also known as SQL packages)
- ▶ COM+, including Microsoft Distributed Transaction Coordinator (DTC) and Microsoft Transaction Services (MTS)
- ▶ Distributed relational database architecture (DRDA), including the CONNECT and DISCONNECT statements

Note: Some of these features are also supported by the DB2 for LUW .NET provider.

As an example, you could take advantage of the extended dynamic SQL support with the ODBC driver by setting appropriate connection properties using the connection keywords. However, before we illustrate the use of the extended dynamic SQL with a coding example, we offer an explanation of why, under certain circumstances, this particular feature might be important for good SQL performance. The SQL packages are server-side repositories for SQL statements. Packages contain the internal structures (such as parse trees and access plans) that are needed to execute SQL statements. Because SQL packages are a shared resource, the information built when a statement is prepared is available to all users of the package. This saves processing time, especially in an environment where many users are using the same or similar statements. Because SQL packages are permanent, this information is also saved across job initiation or termination and across IPLs. In fact, SQL packages can be saved and restored onto other systems.

Example 6-1 shows sample use of ODBC driver–specific keywords to set a database connection that enables extended dynamic SQL support. The SampleOdbc program, written in VisualBasic.NET, is used to read the rows from the STAFF table and display them on the console. Then one of the retrieved rows is updated and the data is redisplayed on the console.

Example 6-1 Sample ODBC .NET provider application

```
Imports System
Imports System.Data
Imports System.Data.OleDb
Imports Microsoft.VisualBasic
[1] Imports Microsoft.Data.Odbc

Public Class Form1

    Public Shared Sub Main()

        Dim strConnectionString As String
        strConnectionString = _
[2]         "DSN=myiSeries;UID=myuserid;PWD=mypassword;DBQ=SAMPLEDB;EXTCOLINFO=1;"
        strConnectionString = strConnectionString & _
[3]         "ExtendedDynamic=1;DefaultPkgLibrary=myschema;"
        Dim pConn As New OdbcConnection(strConnectionString)
        Dim pInsertQuery As String = _
            "SELECT id, name, job, salary FROM Staff FOR UPDATE"
        Dim adapter As New OdbcDataAdapter(pInsertQuery, pConn)
        Dim custCB As OdbcCommandBuilder = New OdbcCommandBuilder(adapter)

        Dim ds As DataSet = New DataSet()
        adapter.Fill(ds, "Staff")
        Console.WriteLine("-----")
```

```

Console.WriteLine("DataSet contents after Fill:")
printDataSet(ds)

Console.WriteLine("-----")
ds.Tables("Staff").Rows(1)("NAME") = "Joanna"
[4] adapter.UpdateCommand = New _
    OdbcCommand("UPDATE Staff SET Name='Joanna' WHERE ID=20",pConn)
adapter.Update(ds, "Staff")

Console.WriteLine("-----")
Console.WriteLine("DataSet contents after Update:")
printDataSet(ds)
Console.WriteLine("-----")

End Sub

Public Shared Sub printDataSet(ds As DataSet)
    Dim table As DataTable
    Dim col As DataColumn
    Dim row As DataRow
    Dim i As Integer

    For Each table in ds.Tables
        For Each col in table.Columns
            Console.Write(col.ColumnName & vbTab & vbTab)
        Next
        Console.WriteLine()

        For Each row in table.Rows
            For i = 0 To table.Columns.Count - 1
                Console.Write(row(i).ToString() & vbTab & vbTab)
            Next
            Console.WriteLine()
        Next
    Next
End Sub

End Class

```

At [1] the ODBC bridge assembly is imported so that the sample application can use unqualified names of the provider classes.

At [2] an OdbcConnection object is created to open a database connection to the iSeries server. Note that the connection string contains parameters that are typical for an ODBC driver. The DSN property is used to point to the target iSeries data source. The DSN must be previously registered with the ODBC Administration utility. The alternate solution is to use the DSN-less syntax for the connection string that eliminates the need for the DSN setup. Here is an example of such a connection string:

```
DRIVER=Client Access ODBC Driver (32-bit);SYSTEM=myiSeries;UID=myUserId;PWD=myPassword;
```

In addition to the required parameters such as DSN, UID, and PWD, we also specify the optional EXTCOLINFO parameter. Setting that property causes the ODBC driver to retrieve additional metadata information such as the base table name for a view. Although not required for our simple SELECT statement, this property may be necessary for more complex queries that involve joins. Setting this parameter ensures that the OdbcCommandBuilder will work properly. Note, however, that EXTCOLINFO causes an additional data flow between the database server and the ODBC driver, which can slow down the performance of the solution.

At [3] the `ExtendedDynamic` keyword specifies whether to use extended dynamic (package) support. A value of 1 enables the packages. This is the default for the iSeries Access for Windows ODBC driver. The `DefaultPkgLibrary` specifies the library for the SQL package. You can omit the `DefaultPkgLibrary`, in which case the package will be created in the default library, QGPL.

6.2 OLE DB .NET Data Provider

The V5R3 release of iSeries Access for Windows includes three OLE DB providers:

- ▶ New SQL-only provider (IBMDASQL) that supports SQL Commitment Control and MTS.
- ▶ New RLA-only provider (IBMDARLA) that supports true record blocking and forward-only cursors for record level access.
- ▶ The IBMDA400 provider, which supports SQL statements, record level access, remote command and program call, and data queues. It does not support Commitment Control for SQL commands, and does not support MTS.

Note: The three OLE DB providers are packaged into a single DLL file.

The `OleDb` .NET Data Provider uses native OLE DB through a COM interop module to enable data access. This provider is a bridge that handles calls from .NET into a traditional COM-style OLE DB Provider. Although .NET applications can connect to the three iSeries Access for Windows OLE DB providers, testing for the Data Queue, Remote Command and Program Call, and Record Level Access support has not been done using the `OleDb` bridge to IBMDA400 or IBMDARLA.

Similar to the ODBC bridge, the OLE DB .NET provider involves jumping in and out of the .NET Framework environment for every interface call because, from the .NET point of view, the iSeries OLE DB provider constitutes unmanaged code.

When running SQL statements and stored procedures, the iSeries OLE DB providers communicate with a back-end database server job called QZDASOINIT by using an iSeries protocol that is highly optimized for performance and functionality.

Note: When using a secure connection, the iSeries database server job is called QZDASSINIT.

Figure 6-2 on page 230 illustrates the software components that are involved in connecting a .NET application to the iSeries database through the OLE DB bridge.

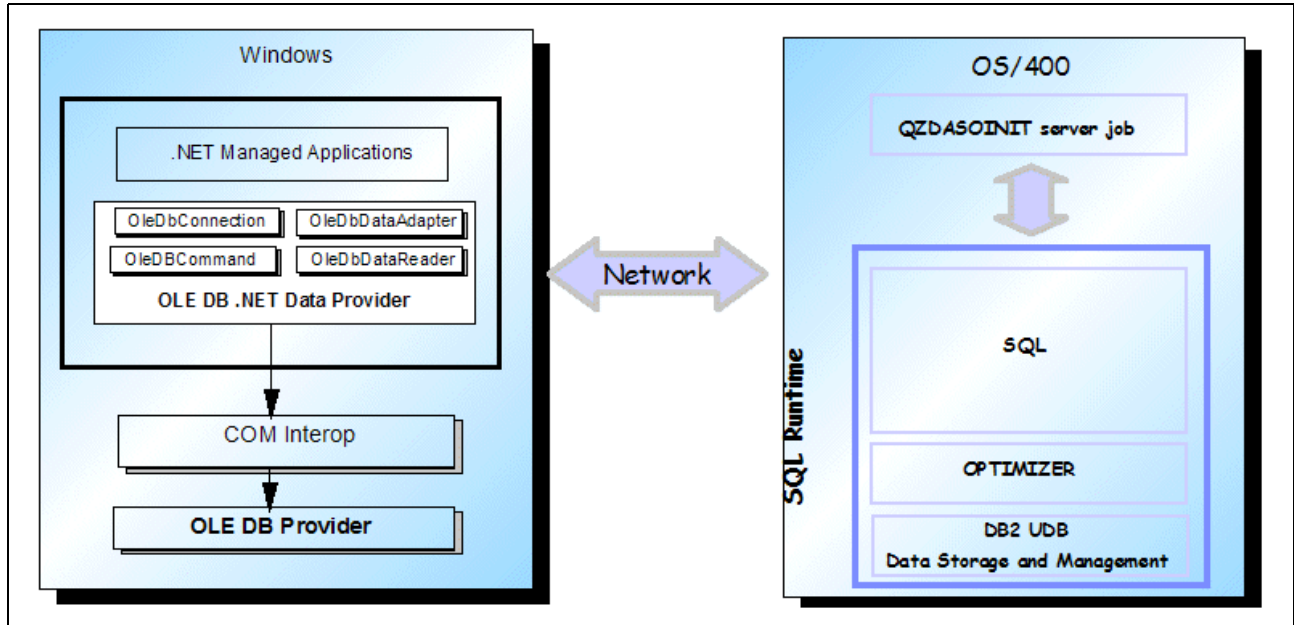


Figure 6-2 OLE DB bridge

The OLE DB bridge can be considered in these cases, where a .NET application requires provider functionality that is not implemented in the DB2 UDB for iSeries .NET provider. Additional features that are available through iSeries OLE DB providers include:

- ▶ Datalink data type.
- ▶ SQL packages.
- ▶ Record level access; you can use either the IBMDA400 or the IBMDARLA OLE DB provider (see **Note**).
- ▶ Data queues; you can use the IBMDA400 OLE DB provider (see **Note**).
- ▶ Remote command and program call; you can use the IBMDA400 OLE DB provider (see **Note**).
- ▶ COM+, including Microsoft Distributed Transaction Coordinator (DTC) and Microsoft Transaction Services (MTS). To use Microsoft distributed transaction coordinator, you can use the System.Data.OleDb .NET provider to bridge to the IBMDASQL OLE DB provider.

Note: Although some features may work, record level access, data queues, and remote command and program call have not been tested through the OleDb bridge. Some of these features are also supported by the DB2 for LUW .NET provider.

So, for example, you could use the IBMDASQL provider to participate in a distributed transaction monitored by COM+ and DTC.

6.3 Provider performance

As we discuss in previous sections, the OLE DB and ODBC bridge providers use the services of native OLE DB and ODBC drivers respectively. Both drivers are unmanaged code, so the communication between a bridge and a driver involves jumping in and out of the .NET framework. In this section we provide performance data collected for a simple .NET

application that uses various .NET providers. The data should aid you in the process of selecting a provider that will meet the performance objectives for your .NET application.

Attention: The performance information in this document is only for guidance. System performance depends on many factors, including system hardware, system and user software, and user application characteristics. Customer applications must be evaluated carefully before estimating performance. IBM does not warrant or represent that a user can or will achieve a similar performance. No warranty on system performance or price/performance is expressed or implied in this document.

For each of the test runs, the sample application uses one of the four supported providers to execute several basic SQL statements that retrieve and manipulate data in a single table. To eliminate the effect of data caching that the iSeries performs automatically, a sample table called COFFEES is recreated before each run. Example 6-2 shows the DDL script used to create the table.

Example 6-2 DDL to create a sample table

```
DROP TABLE COFFEES;
CREATE TABLE COFFEES (
    ID INTEGER GENERATED ALWAYS AS IDENTITY (
        START WITH 1 INCREMENT BY 1
        NO MINVALUE NO MAXVALUE
        NO CYCLE NO ORDER
        CACHE 20 ),
    COF_NAME VARCHAR(32) CCSID 37 DEFAULT NULL ,
    SUP_ID INTEGER DEFAULT NULL ,
    PRICE DOUBLE PRECISION DEFAULT NULL ,
    SALES INTEGER DEFAULT NULL ,
    TOTAL INTEGER DEFAULT NULL ,
    CONSTRAINT MYSCHEMA.COFFEES_PK PRIMARY KEY( ID ) ) ;
COMMIT;
```

The sample application measures the elapsed time for the following database operations:

- **INSERT:** We use the `DataRow` method on a `DataTable` object to create a `DataRow` object. The object is populated with sample data and added to the `DataRowCollection`. We add 5,000 new rows. The new rows are then inserted into the database through a `DataAdapter`. Note that the measured elapsed time for the insert operation is scoped to the `Update` method on the `DataAdapter` object. So, in fact, we measure the efficiency of the .NET provider. This is illustrated in Example 6-3.

Example 6-3 Inserting rows through a DataAdapter

```
For i = 1 To batchSize
    drNewCoffee = dsCoffees.COFFEES.NewRow()
    drNewCoffee.BeginEdit()
    drNewCoffee("COF_NAME") = "Super_Kona_" & i
    drNewCoffee("SUP_ID") = 150
    drNewCoffee("PRICE") = 9.95
    drNewCoffee("SALES") = 1000
    drNewCoffee("TOTAL") = 1000
    drNewCoffee.EndEdit()
    dsCoffees.COFFEES.Rows.Add(drNewCoffee)
Next i
Try
    start_time = Now
    daCoffees.Update(dsCoffees.COFFEES)
```

```

        stop_time = Now
        elapsed_time = stop_time.Subtract(start_time)
    Catch Xcp As Exception

...
End Try

```

- **UPDATE:** We update all 5,000 rows inserted in the previous operation using the DataRow class to modify the COF_NAME column. The Update method on the DataAdapter class is used to propagate the changes to the back-end database. This is shown in Example 6-4.

Example 6-4 Update through a DataAdapter

```

For Each drCoffee In dsCoffees.COFFEES.Rows
    drCoffee("COF_NAME") = "Colombian Select Extra" & drCoffee("ID")
Next
Try
    start_time = Now
    daCoffees.Update(dsCoffees.COFFEES)
    stop_time = Now
    elapsed_time = stop_time.Subtract(start_time)
Catch Xcp As Exception
...
End Try

```

- **Fast DELETE:** We use the ExecuteNonQuery method on the Command object to delete all rows in the table. At V5R3, the performance of a DELETE statement that has no WHERE clause has been greatly improved. If the statement is executed under no commitment control, the system performs the native CLRPFM operation, which is very fast. For statements that run under commitment control, the system performs a specialized CHGPF command. In our case we use DELETE FROM COFFEES WITH NC, which instructs the system to execute the statement with no commitment control. The WITH NC clause in the statement means No Commit (a proprietary iSeries isolation level). Example 6-5 illustrates the relevant code sample.

Example 6-5 Fast delete using a Command object

```

Try
    cmdFastDelete.CommandText = "DELETE FROM COFFEES WITH NC"
    coniSeries.Open()
    start_time = Now
    cmdFastDelete.ExecuteNonQuery()
    stop_time = Now
    coniSeries.Close()
    elapsed_time = stop_time.Subtract(start_time)
Catch Xcp As OdbcException
...
End Try

```

- **SELECT:** We use the Fill method on the DataAdapter object (Example 6-6 on page 233). To avoid the effects of data caching, we actually exit the application and restart it before measuring the performance of the SELECT statement; this releases the database connection that is implicitly pooled by the provider. That in turn recycles the back-end QZDASOINIT job that is associated with the connection. Note that with the IBM DB2 UDB for iSeries .NET provider, we could use the Pooling=false connection property to disable the connection pooling.

Example 6-6 Select using a DataAdapter object

```
Try
    start_time = Now
    dsCoffees.Clear()
    daCoffees.Fill(dsCoffees.COFFEES)
    stop_time = Now
    elapsed_time = stop_time.Subtract(start_time)
Catch Xcp As Exception
    ...
End Try
```

- **DELETE:** We again delete all rows but this time we use the Delete method on the DataRow object to mark all rows in the Table object as deleted. Then we use the Update method on the DataAdapter object to propagate the changes to DB2 (Example 6-7).

Example 6-7 Delete all rows using a DataAdapter object

```
For Each drCoffee In dsCoffees.COFFEES.Rows
    drCoffee.Delete()
Next
Try
    start_time = Now
    daCoffees.Update(dsCoffees.COFFEES)
    stop_time = Now
    conSeries.Close()
    elapsed_time = stop_time.Subtract(start_time)
Catch Xcp As Exception
    ...
End Try
```

So, after discussing the sample code and the methodology that are used to capture the basic performance data, we present the results. Keep in mind that your results may vary from the data shown in this section, as we did not make specialized changes: All performance-related settings were left at their default values. For example, the SELECT performance for the ODBC driver could be improved by increasing the BlockSize property to a larger value, such as 128 KB. (32 KB is the default.) Similarly, the performance of the DB2 LUW .NET provider would be much better if we used the chaining feature implemented by this provider. With chaining enabled, the insert, update, and delete commands executed through a connection are queued on the client. This method is useful for executing a large batch of insert, update, and delete commands because it minimizes the network flow to the server. Note, however, that chaining is *not* supported by the DB2DataAdapter object so we would have to rewrite our application to take advantage of this functionality.

Furthermore, all tests involve just one client accessing a dedicated system. Of course, this is not a typical iSeries production environment. The iSeries performance and scalability will shine in environments with hundreds or thousands of concurrent users running a wide spectrum of business applications. We provide this data to give you a better understanding of how to choose a provider which is right for your application. Table 6-1 on page 234 shows the performance measurements.

Table 6-1 Single user performance test for various .NET providers (results shown in seconds)

	DB2 for iSeries .NET provider	DB2 LUW .NET provider	ODBC “bridge” provider	ILE DB “bridge” provider over IBMDASQL
INSERT	13.81	18.23	26.32	14.39
UPDATE	14.14	17.92	28.21	14.76
SELECT	0.3	0.26	0.39	0.55
FAST DELETE	0.11	1.00	0.1	0.1
DELETE	11.82	18.31	19.57	10.39

Note: Performance comparisons using the DB2 for LUW .NET provider have not been done. The performance should be comparable to that of the DB2 for iSeries .NET provider.

6.4 Conclusions

Choosing the appropriate .NET data provider for your application depends on your environment. If you plan to access only the iSeries database, the DB2 UDB for iSeries .NET provider should be your choice. This managed provider provides better performance than using the System.Data.OleDb provider to bridge to the iSeries Access OLE DB provider, or using the Microsoft.Data.Odbc provider to bridge to the iSeries Access ODBC driver.

Consider the DB2 UDB (LUW) provider if you need to access different DB2 UDB platforms such as DB2 UDB for Windows or DB2 UDB for z/OS in addition to iSeries, or if you wish to take advantage of the GUI Add-Ins we discuss in Chapter 5, “IBM DB2 for LUW .NET provider” on page 177. Using this provider should also be considered if you want to take advantage of the functionality that currently is not supported by the DB2 UDB for iSeries .NET provider, such as distributed transactions managed by MTS/COM+.

Consider using one of the “bridge” providers, if you developed your application for heterogeneous environments in which some of the target databases may not have a managed .NET provider.



Part 3

Scenarios

In this part we illustrate some possible scenarios, such as the use of ASP .NET Web forms.



ASP .NET scenario (Web forms)

An ASP .NET page or a Web form can be connected to the database using ADO .NET and a database provider. In Chapter 2, “Introduction to the Microsoft .NET framework” on page 11 we discussed ADO .NET, and in Chapters 4, 5, and 6 we discuss various providers. In this chapter we discuss different techniques for accessing DB2 UDB for iSeries databases from an ASP .NET Web application. In particular we discuss:

- ▶ An overview of ASP .NET
- ▶ An example demonstrating how to use the IBM DB2 UDB for iSeries .NET provider from an ASP .NET Web page
- ▶ An example demonstrating how to use the IBM DB2 for LUW .NET provider from an ASP .NET Web page
- ▶ Recommendations for using providers

7.1 An overview of ASP.NET

ASP (Active Server Pages) .NET is a part of the Microsoft .NET framework; it is used for making dynamic and interactive Web pages. ASP .NET is the next generation of ASP but it is not an upgraded version of ASP nor is it fully backward compatible with ASP.

ASP .NET provides following features:

- ▶ Increased performance by running compiled code
- ▶ A large set of new programmable controls and XML-based components
- ▶ Event-driven programming
- ▶ Better user authentication
- ▶ Higher scalability
- ▶ Easier configuration and deployment

Moreover, ASP .NET derives its features from .NET Framework; these include support for many languages, ADO .NET support, exception management, garbage collection, Common Type System (CTS), and many more features we discuss in Chapter 1, “Introduction to DB2 UDB for iSeries” on page 3.

7.1.1 ASP .NET Web page (Web form)

The ASP .NET Web page, often called Web form, interacts with users with the help of Web controls. During design time a Web form is stored in a file with the .aspx extension. An ASP .NET Web page can be written using either of these models:

- ▶ In-line model
- ▶ Code-behind model

In-line model

In this model, the code is directly embedded within the ASP .NET page. Here the code is written in <script> blocks in the same .aspx file that contains the HTML and controls.

When the page is deployed, the source code is deployed along with the Web forms page, because it is physically in the .aspx file. However, you do not see the code; only the results are rendered in HTML form when the page runs.

In-line code and DB2 providers

When using the in-line coding method, you must modify your application's web.config file or the system-wide machine.config file to add an assembly reference to the IBM DB2 UDB for iSeries provider or the IBM DB2 for LUW .NET provider.

For example, use the following statement to add a reference to the IBM.Data.DB2.iSeries provider:

```
<add assembly="IBM.Data.DB2.iSeries, Version=10.0.0.0, Culture=neutral,
PublicKeyToken=9cdb2ebfb1f93a26"/>
```

Tip: The PublicKeyToken listed here can be found by displaying the .NET Global Assembly Cache (GAC).

In addition to adding the assembly reference to the .config file, you should add an Import statement for the .NET provider to your Web form (for example, the .aspx file) to avoid having to fully qualify each use of a .NET provider class name with the IBM.Data.DB2.iSeries or IBM.Data.DB2 prefix.

An example of the Import statement for IBM.Data.DB2.iSeries provider is given below:

```
<%@ Import Namespace="IBM.Data.DB2.iSeries" %>
```

Applications that use the in-line code model but do not add the proper assembly references encounter the following error:

Compilation Error

Description: An error occurred during the compilation of a resource required to service this request. Review the following specific error details, and modify your source code appropriately.

Compiler Error Message: BC30002: Type 'iDB2Connection' is not defined.

Code-behind model

In this model, the HTML and the controls are in the .aspx file, and the code is written in a separate .aspx.vb or .aspx.cs file. The code-behind model keeps your business logic separate from your Web page design, and in general is easier to write and maintain.

All project class files (without the .aspx file itself) are compiled into an assembly (.dll file), which is deployed on the server without any source code. When a request for the page is received, then an instance of the .dll file is created and executed on the Web server.

Code-behind and DB2 providers

When using the code-behind model with a DB2 provider, you must add a project reference to the IBM DB2 UDB for iSeries provider (IBM.Data.DB2.iSeries) or the IBM DB2 for LUW .NET provider (IBM.Data.DB2). In addition, you should also add a using (C#) or Imports (VB .NET) statement to your source files (for example, .cs or .vb files) to avoid having to fully qualify each use of a .NET provider class name with the provider prefix.

We discuss adding a project reference to the providers in 4.4.3, “Adding an assembly reference to the provider” on page 43 and in 5.6.3, “Adding an assembly reference to the provider” on page 198.

7.1.2 How does ASP .NET work?

By default, Visual Studio .NET creates ASP .NET Web pages using the code-behind model and compiles it into an assembly or .dll file. An assembly is deployed and configured on IIS to become a Web application.

Figure 7-1 on page 240 shows how an ASP .NET Web page is processed on the Web server.

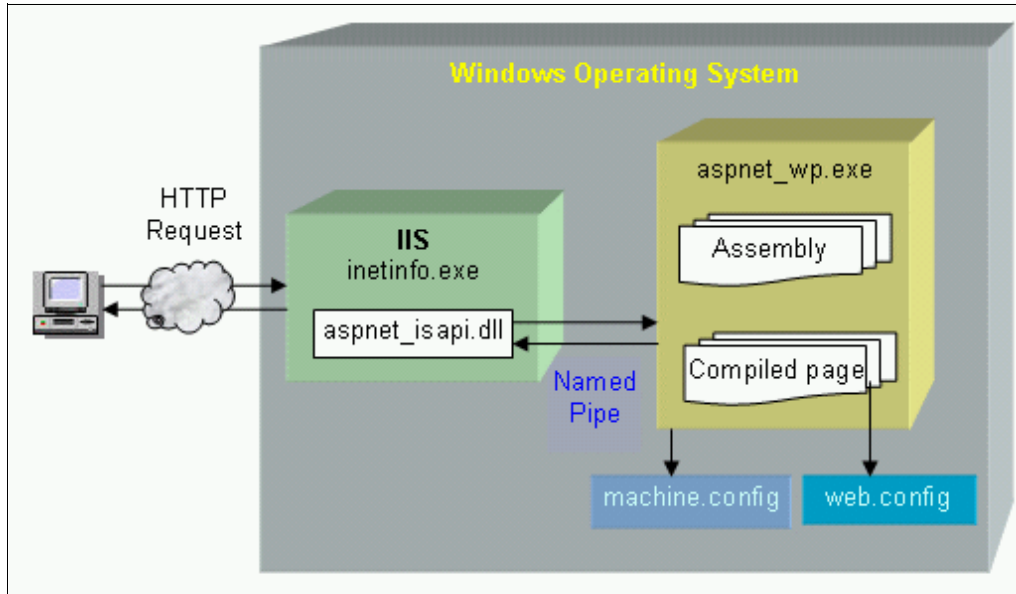


Figure 7-1 How ASP.NET works

When the user navigates to one of the Web forms (<http://www.myweb.com/default.aspx>, for example), the following sequence occurs:

1. IIS starts the ASP .NET worker process (aspnet_wp.exe) if it is not already running. The ASP .NET worker process loads the assembly associated with the Web form by allocating one process space, called the application domain. The application domain is created for each application and provides isolation between processes.
2. The assembly composes a response to the user based on the content of the Web form that the user requested and any program logic that provides dynamic content.
3. IIS returns the response to the user in the form of HTML.

7.1.3 Configuration files in ASP .NET

In .NET, configuration files are XML files that can be changed as needed without having to recompile applications. The configuration file can be used to store your connection string, user ID, or password. The .NET Framework has three basic configuration files:

► Application configuration file

In ASP .NET, the application configuration file is known as web.config and can be used to store Web application–related settings, such as connection string. At run time, ASP .NET uses the configuration information provided by the web.config file.

► Machine configuration file

The machine.config file has all of the settings for the machine and can be found in the CONFIG directory of the runtime install path.

The machine.config file is useful for storing connection strings, if there is more than one Web application using a common connection string.

► Security configuration file

This file contains information about the code group hierarchy and permission sets associated with a policy level.

7.2 Web controls

Web controls, an essential part of any ASP .NET Web page, act as user interface (UI) elements. Web controls are classified in different ways:

- ▶ Built-in controls
 - HTML controls
 - Web form controls
 - Field validator controls
- ▶ Custom controls

Built-in controls

Built-in controls are available and included with Visual Studio .NET. The next section describes available built-in controls in the .NET Framework.

HTML controls

These traditional HTML controls are represented using HTML tags. In Visual Studio .NET you can drag and drop HTML controls or you can directly write HTML script for displaying the controls. These are client-side controls that process the code using script language.

Web form controls

ASP .NET is popular because of the variety of Web form controls. Due to the Web form controls, designing and programming Web pages is simple and fast. Web form controls are created and run on the server. After processing at the server, they render the output into the appropriate HTML code for sending it to the client. All Web form controls inherit from a common base class: `System.Web.UI.WebControls`.

Table 7-1 shows some of the important Web controls and their descriptions.

Table 7-1 Web form controls

Web control	Description
Label	Can be used to display read-only text on the HTML page.
TextBox	Can be used to display an input area on the HTML form.
Button, LinkButton, ImageButton	These are various types of buttons differentiated according to their appearance. When clicked, they post command information back to the server.
Hyperlink	A hyperlink control that responds to a click event.
DropDownList, ListBox	Behaves the same as HTML controls but can be bound to a data source.
DataGrid	A powerful Web control to display data, and which has features such as paging, sorting, and formatting. Can also be used for various other operations (editing, deleting).
DataList	Used to display data in a non-tabular format. Requires more coding for design layout using templates. Supports editing and deleting data.
Repeater	This control is not derived from the WebControl class, and therefore does not have the stylistic properties. Does not have built-in functionality for editing and deleting data.

Web control	Description
CheckBox, CheckBoxList	Displays a check box or a group of check boxes. Can be bound and used to display Boolean data.
RadioButton, RadioButtonList	Displays a radio button or a group of radio buttons. Can be bound and used to display Boolean data.
Image	Displays an image within the page.
Calendar	Creates an HTML version of a pre-settable calendar.

Field validator controls

Field validator controls are used to validate the data on the client browser before the user submits the data to the back-end server. These controls automatically write client-side JavaScript code into the HTML page so the values can be checked without the round-trip to server.

Some of the field validator controls are:

- ▶ RequiredFieldValidator
- ▶ CompareValidator
- ▶ RangeValidator
- ▶ RegularExpressionValidator
- ▶ CustomValidator

Custom controls

In addition to the built-in controls, you can also build your own custom controls in the .NET Framework.

7.3 Using the IBM DB2 UDB for iSeries .NET provider

In Chapter 4, “IBM DB2 UDB for iSeries .NET provider” on page 33, we discuss the IBM DB2 UDB for iSeries .NET provider in detail. In this section we show how to use the IBM DB2 UDB for iSeries .NET provider in conjunction with a Web form.

We assume that you are developing ASP .NET application using Visual Studio .NET 1.1 and have the IBM DB2 iSeries provider properly installed on your server machine. It is always a good practice to check the connectivity between your development environment and the iSeries before you start programming. (See 4.2.2, “Host server jobs” on page 35.) Be sure that IIS is running properly in the development environment where you are building your Web application.

To start, open Visual Studio .NET and select **File** → **New** → **Project** to open the New Project window (Figure 7-2 on page 243). Under Project Types, select **Visual C# Projects**, and in the Templates pane, select **ASP .NET Web Application**. Specify a location, such as the Web server name (localhost) and an application name (iSeriesWeb).

Click **OK**, and Visual Studio .NET automatically creates a Web application on the specified Web server with a default configuration file and a Web page.

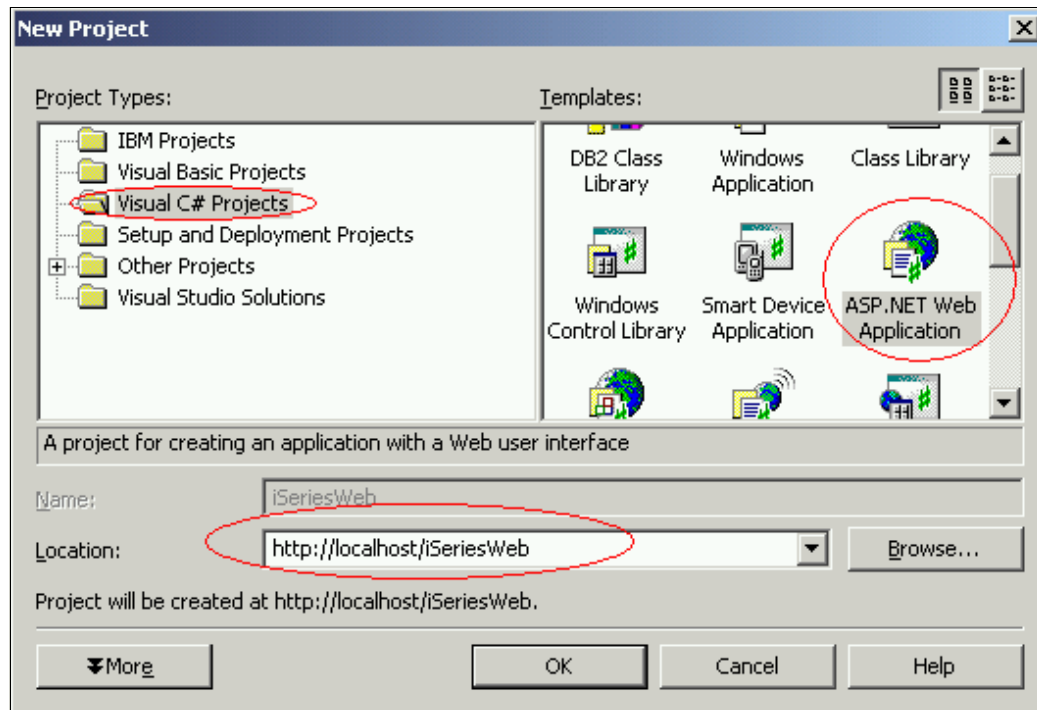


Figure 7-2 New Project window

In Solution Explorer, rename the default WebForm1.aspx Web page to iSeries.aspx (Figure 7-3).

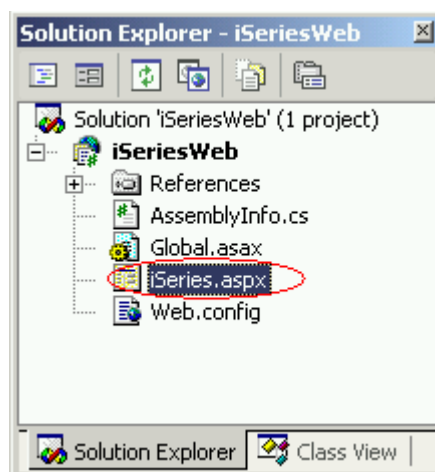


Figure 7-3 Solution Explorer: renaming WebForm1.aspx to iSeries.aspx

To access the database on the iSeries, you must add a reference to the IBM.Data.DB2.iSeries provider. Right-click the project name (**iSeriesWeb**) in Solution Explorer and select **Add Reference** to open the Add Reference window (Figure 7-4 on page 244).

In the Add Reference window, select **IBM DB2 UDB for iSeries .NET Provider** and click **Select** to add the provider to the Selected Components list. Click **OK** to add the reference to the iSeries provider.

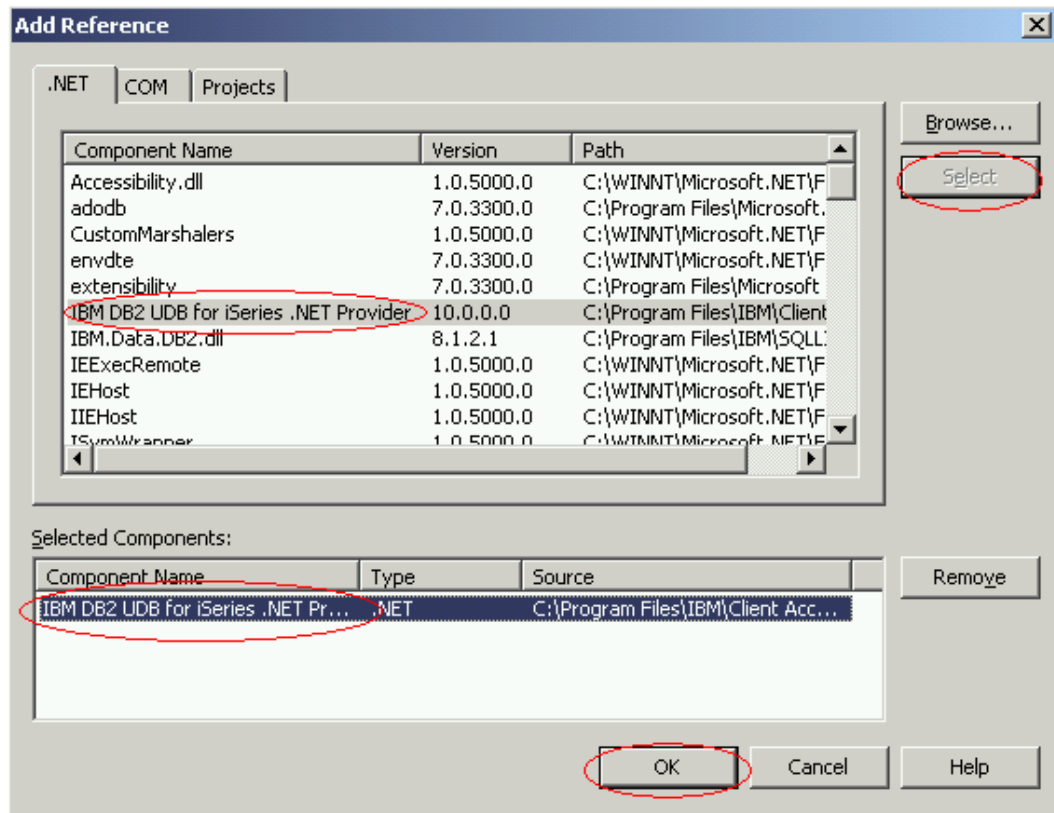


Figure 7-4 Reference window: adding a reference to the IBM DB2 UDB for iSeries .NET provider

Alternatively, you can verify the added references in the project by expanding the References node in your Solution Explorer window, as shown in Figure 7-5.

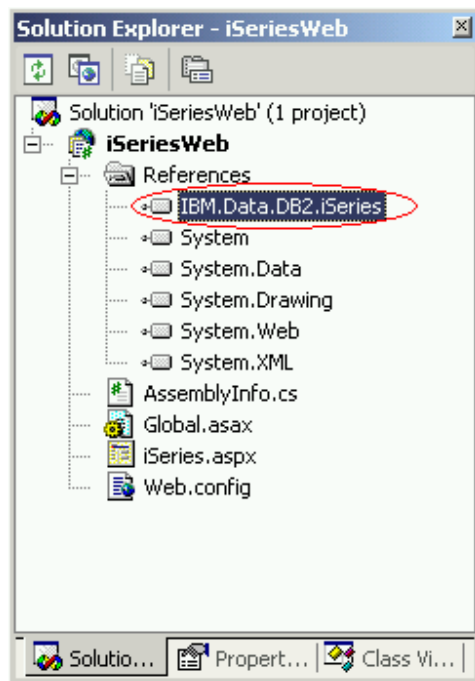


Figure 7-5 Solution Explorer: references in the project

In this application, we show how to populate a ListBox control and DataGrid control using the IBM DB2 UDB for iSeries .NET provider. We use the ListBox control to populate the First Name column from the EMPLOYEE table and use the DataGrid to display the STAFF table.

To design the user interface, double-click or drag the controls from the Web Forms toolbox on the iSeries.aspx design page, and change the text for Button1 to Fill List Box and the text for Button2 to Fill Data Grid as shown in Figure 7-6.

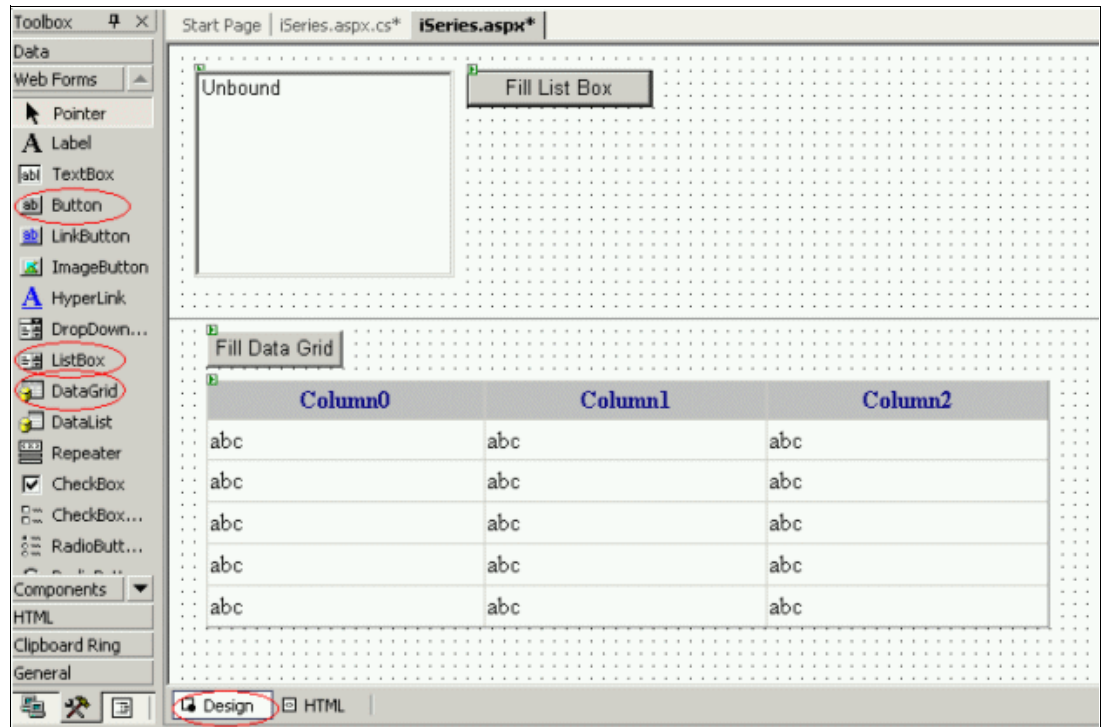


Figure 7-6 Web page design: user interfaces

You can apply various designs and styles to the DataGrid control by right-clicking it and selecting **Property Builder**.

In the next section, we see the actual code that is used to populate the controls with data from the iSeries database.

Populating the ListBox

To add code for populating the ListBox control, double-click **Fill List Box**. Visual Studio .NET automatically opens the code window with code for the button's click event (Example 7-1).

Example 7-1 Code for click event of Button1

```
private void Button1_Click(object sender, System.EventArgs e)
{
```

Create a connection using an iDB2Connection object (Example 7-2 on page 246).

Important: When using the provider through ASP .NET / IIS, the connection string should *always* contain the authentication credentials, such as the UserID and Password elements. Otherwise a logon window may appear at the server side, which would make the Web client appear to hang.

Example 7-2 Code for the click event of Button1: creating the connection

```
try
{
    iDB2Connection connDB2=new iDB2Connection(
        "DataSource=myiSeries;"
        +"userid=myuserid;password=mypassword;DefaultCollection=sampledb;");
}
```

Create an iDB2Command object using a valid SQL statement and instance of connection object, then open the connection (Example 7-3).

Example 7-3 Code for the click event of Button1: creating the command

```
iDB2Command idbc = new iDB2Command("select FIRSTNAME, LASTNAME from employee",CommandType.Text,
connDB2);
connDB2.Open ();
```

After opening the connection, execute the command object using the ExecuteReader () method and assign it to the DataSource property of the ListBox control (Example 7-4).

Example 7-4 Code for the click event of Button1: executing the command

```
ListBox1.Items.Clear();
ListBox1.DataSource = idbc.ExecuteReader();
```

Assign the DataTextField and DataValueField properties of the list box with the appropriate column names (Example 7-5), and bind the ListBox using the DataBind() method. The DataTextField property specifies a field in the DataSource to display as the items of the list in a list control, and the DataValueField specifies a field that contains the value of each item in a list control. You can use a client-side script to get the DataValueField of the selected list item.

Example 7-5 Code for the click event of Button1: binding the control

```
ListBox1.DataTextField = "FIRSTNAME" ;
ListBox1.DataValueField = "LASTNAME";

ListBox1.DataBind();
```

Close the connection using the Close() method (Example 7-6).

Example 7-6 Code for the click event of Button1: closing the connection

```
connDB2.Close();
}
catch (Exception eR)
{
    Response.Write (eR.Message.ToString());
}
}
```

Example 7-7 shows the entire code for populating the ListBox control.

Example 7-7 Complete code for the click event of Button1

```
private void Button1_Click(object sender, System.EventArgs e)
{
    try
    {
        iDB2Connection connDB2=new iDB2Connection(
            "DataSource=myiSeries;" +
            "userid=myuserid;password=mypassword;DefaultCollection=sampled;" );

        iDB2Command idbc = new iDB2Command("select FIRSTNAME, LASTNAME from
employee",CommandType.Text, connDB2);
        connDB2.Open ();
        ListBox1.Items.Clear();
        ListBox1.DataSource = idbc.ExecuteReader();

        ListBox1.DataTextField = "FIRSTNAME" ;
        ListBox1.DataValueField = "LASTNAME";

        ListBox1.DataBind();

        connDB2.Close();
    }
    catch (Exception eR)
    {
        Response.Write (eR.Message.ToString());
    }
}
```

Binding the DataGrid

The DataGrid control is a very powerful control for displaying data, and it supports selection, editing, deletion, paging, and sorting features.

In this example we show how to display data in the DataGrid control from the STAFF table. You can add code for binding the DataGrid control by double-clicking the Fill Data Grid button as shown in Example 7-8.

Example 7-8 Code for the click event of Button2 to bind the datagrid

```
private void Button2_Click(object sender, System.EventArgs e)
{
    try
    {
```

The first step in making a connection with iSeries is to create a connection as shown in Example 7-9.

Example 7-9 Code for the click event of Button2 to bind the DataGrid: creating the connection

```
iDB2Connection connDB2=new iDB2Connection(
    "DataSource=myiSeries;" +
    "userid=myuserid;password=mypassword;DefaultCollection=sampled;" );
```

Create the DataSet and iDB2DataAdapter objects. The DataSet object represents an in-memory cache of data from the STAFF table, and the iDB2DataAdapter is created to fill the DataSet and update the data source.

Bind the DataGrid control with the DataSource using the DataBind() method (Example 7-10).

Example 7-10 Code for the click event of Button2 to bind the DataGrid: filling the dataset

```
DataSet ds = new DataSet();
iDB2DataAdapter adpt = new iDB2DataAdapter();

adpt.SelectCommand = new iDB2Command("select * from STAFF", connDB2);
adpt.Fill(ds);

DataGrid1.DataSource = ds;
DataGrid1.DataBind ();
connDB2.Close();
```

Example 7-11 shows the complete example for binding the DataGrid with the database using the DB2 UDB for iSeries provider.

Example 7-11 Complete code for the click event of Button2 to bind the DataGrid

```
private void Button2_Click(object sender, System.EventArgs e)
{
    try
    {
        iDB2Connection connDB2=new iDB2Connection(
            "DataSource=myiSeries;" +
            "userid=myuserid;password=mypassword;DefaultCollection=sampled;" );

        DataSet ds = new DataSet();
        iDB2DataAdapter adpt = new iDB2DataAdapter();
        adpt.SelectCommand = new iDB2Command("select * from staff",
connDB2);
        adpt.Fill(ds);

        DataGrid1.DataSource = ds;
        DataGrid1.DataBind ();
        connDB2.Close();
    }
    catch (Exception e)
    {
        Response.Write (e.Message);
    }
}
```

After clicking both buttons, the output is shown on the iSeries.aspx Web page (Figure 7-7 on page 249).

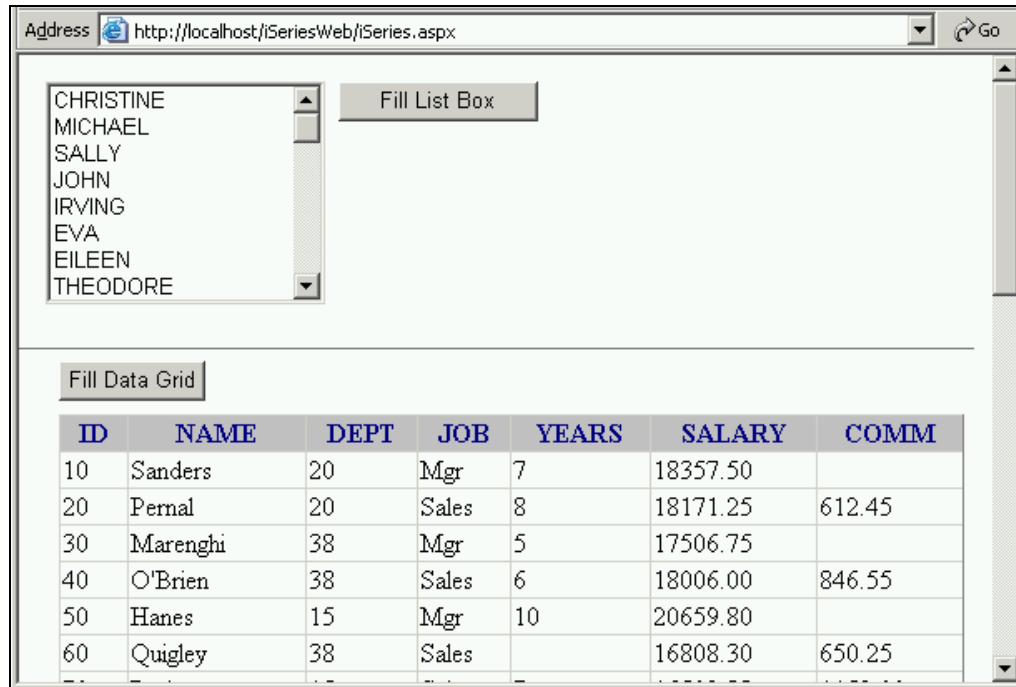


Figure 7-7 Output of iSeries.aspx Web page (using the IBM DB2 UDB for iSeries provider)

7.4 Using the IBM DB2 for LUW .NET provider

In this section we show how to use the IBM DB2 for LUW .NET provider to populate the DataRepeater and DataList Web controls.

We start with the procedure that we discussed in 7.3, "Using the IBM DB2 UDB for iSeries .NET provider" on page 242 and create a new ASP .NET Web application. Rename it to `luwWeb`. Rename the default `WebForm1.aspx` page to `luw.aspx`.

In "Using the IBM DB2 UDB for iSeries .NET provider" on page 242 we discussed how to add a reference to the IBM DB2 UDB for iSeries provider. Follow the same steps, but this time add a reference to the IBM DB2 for LUW provider. The Add References window looks similar to Figure 7-8 on page 250. In this window, select **IBM.Data.DB2.dll** and click **OK**.

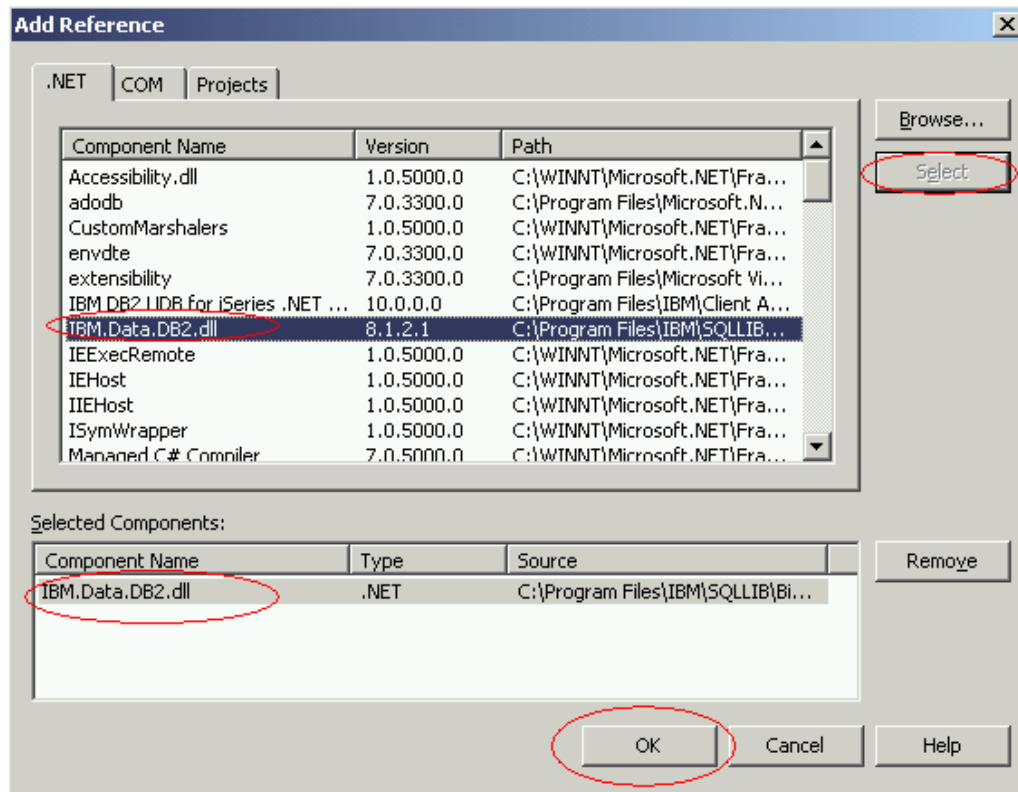


Figure 7-8 Add Reference window: adding reference to IBM DB2 LUW .NET provider

Next, design your Web page by dragging the controls from the Toolbox window and renaming the button text as shown in Figure 7-9.

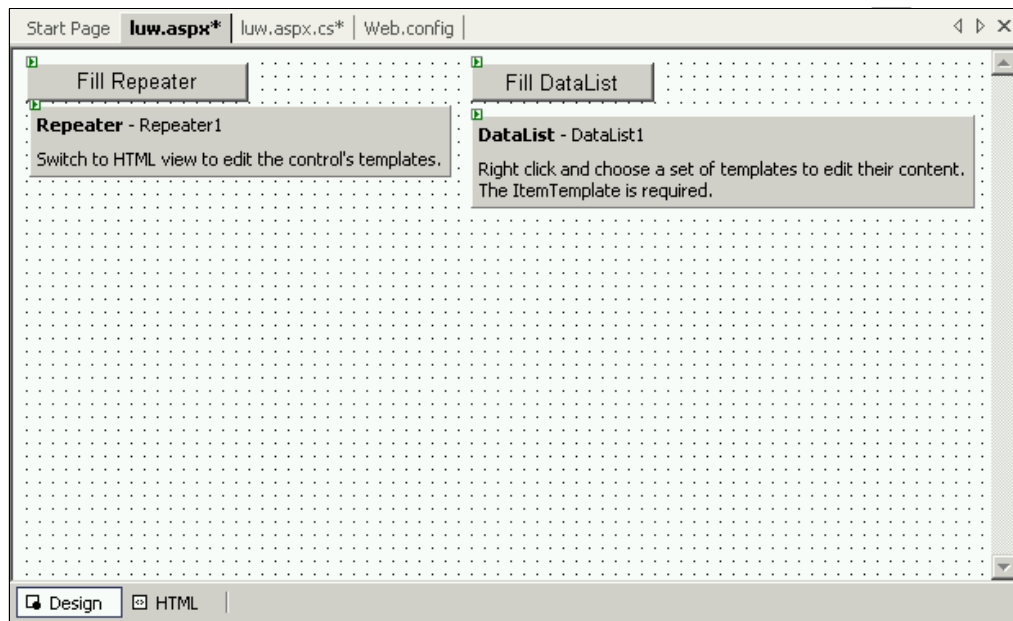


Figure 7-9 Design window

Designing the Repeater control

The Repeater is a good choice for displaying data in a format other than HTML <table>.

Note that the Repeater control is not derived from the WebControl class, so it does not have the stylistic properties (Font, BackColor, BorderStyle) that are common to all Web controls. Also, the Repeater control does not provide any built-in features for editing, sorting, or paging of the data.

To represent the data, the following templates can be used with the Repeater control:

- ▶ AlternatingItemTemplate
- ▶ ItemTemplate
- ▶ HeaderTemplate
- ▶ FooterTemplate
- ▶ SeparatorTemplate

In this example we use HeaderTemplate, ItemTemplate, and FooterTemplate to display the data in tabular form. To add code for templates, go to the HTML code window and add code for the Repeater control as shown in Example 7-12.

Example 7-12 Code for repeater control

```
<asp:repeater id="Repeater2" runat="server">
  <HeaderTemplate>
    <table border="1">
      <tr>
        <td><b>Name of Employee</b></td>
        <td><b>BirthDate</b></td>
        <td><b>Phone No</b></td>
      </tr>
    </table>
  </HeaderTemplate>
  <ItemTemplate>
    <tr>
      <td><%# DataBinder.Eval(Container.DataItem, "FIRSTNAME")%>
        <%# DataBinder.Eval(Container.DataItem, "LASTNAME")%>
      </td>
      <td><%# DataBinder.Eval(Container.DataItem, "BIRTHDATE", "{0:d}")%></td>
      <td><%# DataBinder.Eval(Container.DataItem, "PHONENO")%></td>
    </tr>
  </ItemTemplate>
  <FooterTemplate>
    </table>
  </FooterTemplate>
</asp:repeater>
```

The design page automatically reflects the code that you added in the templates, as shown in Figure 7-10 on page 252.

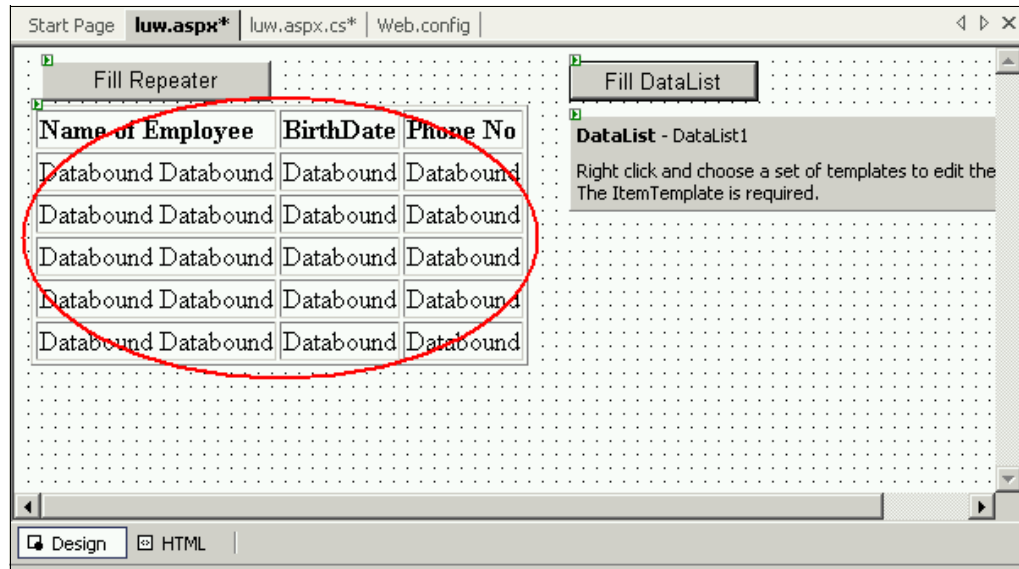


Figure 7-10 Auto-updated Design window after inserting template for Repeater control

Designing the DataList control

The DataList Web control is useful for displaying data that can be highly customized in its layout. The DataList control is capable of performing sorting, paging, and editing of its data, but requires more coding than the DataGrid control.

It uses the following templates to represent the data in a variety of design layouts:

- ▶ AlternatingItemTemplate
- ▶ EditItemTemplate
- ▶ HeaderTemplate
- ▶ FooterTemplate
- ▶ SeparatorTemplate

In this example we use HeaderTemplate and ItemTemplate to lay out the data. To specify template properties, open the HTML code window and enter the code as shown in Example 7-13 for the DataList control.

Example 7-13 Code for DataList control

```
<asp:datalist id="Datalist1" style="Z-INDEX: 102; LEFT: 424px; POSITION: absolute; TOP: 48px"
runat="server" Width="184" Height="88">
<HeaderStyle BackColor="#aaaadd"></HeaderStyle>
<AlternatingItemStyle Back-Color="Gainsboro"></AlternatingItemStyle>
<EditItemStyle BackColor="yellow"></EditItemStyle>
<HeaderTemplate>
Staff and Salary
</HeaderTemplate>
<ItemTemplate>
Staff Name:
<%# DataBinder.Eval(Container.DataItem, "NAME") %>
<br>
Salary:
<%# DataBinder.Eval(Container.DataItem, "SALARY") %>
<br>
</ItemTemplate>
</asp:datalist>
```

The design page automatically reflects the code for the DataList control, which is added using templates and shown in Figure 7-11.

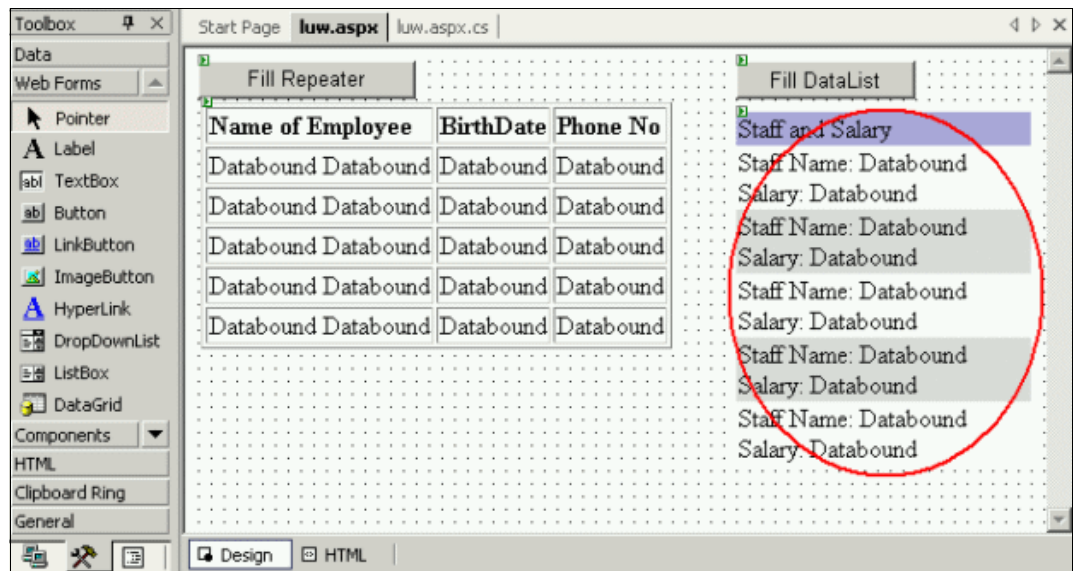


Figure 7-11 Auto-updated Design window after inserting the template for DataList control

Binding the Repeater control

After specifying the templates and design layout, we have to bind it in order to display the actual data. In this example we use the web.config file for declaring the connection string. The connection string can be specified using the appSettings element. To declare the connection string, open the web.config file and add the following code after the </system.web> element:

```
<appSettings>
  <add key="connectionString" value="database=myDB2;Connect Timeout=30; user
  Id=myuserid;password=mypassword;" />
</appSettings>
```

Note: We assume here that the database name myDB2 is already configured and created on your development machine. For more details, see 5.2.4, “Connecting to an iSeries database” on page 180.

Save and close the web.config file and open the Design window for the luw.aspx Web page. Double-click **Fill Repeater** so that Visual Studio .NET automatically adds an event for the button. Add code to create a new connection, as shown in Example 7-14.

Example 7-14 Code for the Fill Repeater button

```
private void Button1_Click(object sender, System.EventArgs e)
{
    try
    {
        DB2Connection connDB2=new DB2Connection ();
```

After creating the instance of the DB2Connection object, specify the connection string declared in the web.config file as shown in Example 7-15.

Example 7-15 Code for Fill Repeater button

```
connDB2.ConnectionString =  
ConfigurationSettings.AppSettings["ConnectionString"];
```

Specify the SQL statement using the DB2Command object and bind the Repeater control. Example 7-16 shows the complete code for using the Repeater control with the IBM DB2 for LUW provider.

Example 7-16 Binding the Repeater control using the IBM DB2 for LUW provider

```
private void Button1_Click(object sender, System.EventArgs e)  
{  
    try  
    {  
        DB2Connection connDB2=new DB2Connection ();  
  
        connDB2.ConnectionString = ConfigurationSet  
tings.AppSettings["ConnectionString"];  
  
        DB2Command cmdDB2=new DB2Command (  
            "SELECT * FROM SAMPLEDB.EMPLOYEE", connDB2);  
  
        connDB2.Open();  
  
        Repeater1.DataSource = cmdDB2.ExecuteReader();  
        Repeater1.DataBind();  
  
        connDB2.Close();  
    }  
    catch (Exception ex)  
    {  
        Response.Write (ex.Message);  
    }  
}
```

Binding the DataList control

The code for binding the DataList control is the same as discussed previously in “Binding the Repeater control” on page 253, except for the SQL statement. In this example we use the STAFF table to display the data in the DataList control.

Example 7-17 shows the complete code for displaying the data using the DataList control.

Example 7-17 Binding the Repeater control using the IBM DB2 for LUW provider

```
private void Button2_Click(object sender, System.EventArgs e)  
{  
    try  
    {  
        DB2Connection connDB2=new  
DB2Connection(ConfigurationSettings.AppSettings["ConnectionS  
tring"]);  
  
        DB2Command cmdDB2=new DB2Command(  
            "SELECT * FROM SAMPLEDB.STAFF", connDB2);  
  
        connDB2.Open();
```

```

        DataList1.DataSource = cmdDB2.ExecuteReader();
        DataList1.DataBind();

        connDB2.Close();
    }
    catch (Exception ex)
    {
        Response.Write (ex.Message);
    }
}

```

After clicking Fill Repeater and Fill DataList, the luw.aspx Web page looks like Figure 7-12.



Figure 7-12 Output of the luw.aspx Web page using the IBM DB2 for LUW provider



A

Sample programs

Throughout much of this book, coding examples are shown using the C# programming language. We also provide downloadable sample programs in both C# and Visual Basic that illustrate many of the examples from this book. This appendix is a reference that explains the contents of each sample program. We have arranged the samples in the order they appear in the book. See Appendix B, “Additional material” on page 261 for information about downloading the samples.

Samples for the IBM DB2 UDB for iSeries .NET provider

This section describes the sample programs for many of the examples shown in Chapter 3, “ADO .NET object hierarchy” on page 17 and Chapter 4, “IBM DB2 UDB for iSeries .NET provider” on page 33. We provide samples in both Visual Basic and in C#.

Sample program	Description	References
AdoNetConnectedMode	Uses a stored procedure to retrieve Employee information	3.2, “Connected mode” on page 23 Example 3-6 through Example 3-9
AdoNetDisconnectedMode	Uses a DataAdapter to read and write Department information	3.3, “Disconnected mode” on page 25 Example 3-10 through Example 3-19
SimpleConnectionExample	Connects to the iSeries and displays the JobName	4.5.1, “A simple connection example” on page 45 Example 4-1
ConnectionPropertiesAndMethods	Shows how to initialize and use iDB2Connection objects	4.5.2, “iDB2Connection and ConnectionString properties” on page 48 Example 4-2 through Example 4-39
CommandPropertiesAndMethods	Shows how to create and use iDB2Command objects	4.5.3, “iDB2Command properties and methods” on page 65 Example 4-41 through Example 4-54
UsingParameters	Shows how to use parameters in your SQL statements	4.5.4, “Using parameters in your SQL statements” on page 74 Example 4-55 through Example 4-61
CallingStoredProcedures	Shows how to call stored procedures, pass parameters, handle result sets, and return values	4.5.5, “Calling stored procedures” on page 79 Example 4-62 through Example 4-68
DataTypeExamples	Shows different ways to handle data types, including char data tagged with CCSID 65535, date and time values, and decimal data	4.5.7, “Provider data types” on page 87 Example 4-69 through Example 4-82
ExceptionHandling	Shows how to handle exceptions in your application	4.5.8, “Handling exceptions” on page 102 Example 4-83 through Example 4-86
DataReaderConsole	Shows how to use a DataReader to read from a table in a Console application	“A Console DataReader example” on page 107 Example 4-87
DataReaderWindow	Shows how to use a DataReader to read from a table in a Windows application	“A Windows DataReader example” on page 108 Example 4-88
DataAdapterWithCommandBuilder	Uses a DataAdapter with CommandBuilder to read from a table	4.6.2, “A simple DataAdapter with CommandBuilder example” on page 110 Example 4-89
UsingTransactions	Shows how to use iDB2Transaction objects to control a transaction	4.6.3, “Using transactions” on page 116 Example 4-90 through Example 4-93

Sample program	Description	References
UsingQCMDEXC	Shows different ways to use the QCMDEXC stored procedure to call i5/OS programs and commands	4.6.5, "Calling a program or CL command using QCMDEXC" on page 120 Example 4-94 through Example 4-101
CultureSpecificSettings	Shows how culture settings can change the way data is handled by the provider	4.7.1, "Internationalization and support for multiple languages" on page 129 Example 4-102 through Example 4-104
LobWindow	Reads LOB data using ExecuteScalar and displays the result in a PictureBox	4.7.2, "Using large objects (LOBs)" on page 132 Example 4-105
LobConsole	Reads LOB data using the GetChars method	4.7.2, "Using large objects (LOBs)" on page 132 Example 4-106
UpdatableDataSet	Uses a DataAdapter to read and write data	4.7.3, "Updating DataSets" on page 136 Example 4-107 through Example 4-109
UsingDataLinks	Shows how to Select and Insert using Datalinks	4.7.5, "Using DataLinks" on page 141 Example 4-110 through Example 4-112
ConnectionPooling	Illustrates how connection pooling works	4.7.6, "Connection pooling" on page 143 Example 4-113 through Example 4-114
AdoNetForwardOnlyReadOnly	Shows an ADO forward-only, read-only example converted to ADO.NET	"Forward-only, read-only recordset example using ADO.NET" on page 155 Example 4-116
AdoNetUpdatableRecordset	Shows an ADO updatable recordset example converted to ADO.NET	"Updatable recordset example using ADO.NET" on page 163 Example 4-118
TryCatchExample	Shows how to handle exceptions using a Try/Catch block	4.10.1, "Handle exceptions using try/catch blocks" on page 166 Example 4-119
ProviderIndependence	Shows different ways to use multiple providers in your application	4.11, "Writing code for provider independence" on page 171 Example 4-122 through Example 4-124

Sample for the IBM DB2 for LUW .NET provider

This is the sample program for the Employee Photo Viewer example shown in Chapter 5, "IBM DB2 for LUW .NET provider" on page 177. This sample is provided in C#.

Sample program	Description	References
EmployeePhotoViewer	Reads Employee pictures as LOB data and displays them in a PictureBox	5.7.1, "Using large objects (LOBs)" on page 204 Example 5-8 through Example 5-14



Additional material

This redbook refers to additional material that can be downloaded from the Internet as described below.

Locating the Web material

The Web material associated with this redbook is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

<ftp://www.redbooks.ibm.com/redbooks/SG246440>

Alternatively, you can go to the IBM Redbooks Web site at:

ibm.com/redbooks

Select **Additional materials** and open the directory that corresponds with the redbook form number, SG246440.

Using the Web material

The additional Web material that accompanies this redbook includes the following file:

<i>File name</i>	<i>Description</i>
SG24-6440-SamplePrograms.zip	Zipped Code Samples

System requirements for downloading the Web material

The following system configuration is recommended:

- ▶ iSeries requirements
 - OS/400 Version 5 Release 3 or later
 - 5722-SS1 Host Servers
 - 5722-XE1 iSeries Access for Windows V5R3

- ▶ PC requirements
 - Windows XP
 - iSeries Access for Windows V5R3
 - Microsoft .NET framework and Microsoft Visual Studio .NET

How to use the Web material

To extract the files, right-click the zip file and select **Extract all**. It will extract the files into a folder named c:\SG24-6440-SamplePrograms on your computer. Inside that folder are three folders:

iDB2Samples-VB	Contains Visual Basic samples for the DB2 UDB for iSeries .NET provider.
iDB2Samples-VC	Contains Visual C# samples for the DB2 UDB for iSeries .NET provider.
DB2Samples-VC	Contains a Visual C# sample for the DB2 for LUW .NET provider.

To use the sample programs, navigate into the appropriate directory and double-click the .sln file. This opens Visual Studio .NET. The iDB2 samples were created using Visual Studio .NET 2002, so if you run them on a later version of Visual Studio .NET, the first time you open the .sln file you will be asked if you want to convert the files to the newer version. Select **Yes**. The DB2 sample was created using Visual Studio .NET 2003.

Each solution contains multiple samples. To work with a particular sample, display the Solution Explorer. Locate the sample you wish to run, then right-click it and select **Set as Startup Project**. Then run the sample.

We recommend that you run each sample by stepping through the examples using Visual Studio .NET in debug mode so you can see what each sample does. The samples may have to be modified to run in your environment. Most of them use the SAMPLEDB database we discuss in Chapter 1. Because the samples were written to be used by anyone, you may have to change at least the ConnectionString used by the samples. For example, in many places where we use the name myiSeries, you should substitute the name of your iSeries server.

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

IBM Redbooks

For information on ordering these publications, see “How to get IBM Redbooks” on page 264. Note that some of the documents referenced here may be available in softcopy only.

- ▶ *Advanced Functions and Administration on DB2 Universal Database for iSeries*, SG24-4249
- ▶ *DB2 UDB V8.2 on the Windows Environment*, SG24-7102
- ▶ *DB2 Universal Database for iSeries Administration: The Graphical Way on V5R3*, SG24-6092
- ▶ *iSeries Access for Windows V5R2 Hot Topics: Tailored Images, Application Administration, SSL, and Kerberos*, SG24-6939
- ▶ *Lotus Domino and .NET coexistence*, REDP-3868
- ▶ *Stored Procedures, Triggers, and User Defined Functions on DB2 Universal Database for iSeries*, SG24-6503
- ▶ *Striving for Optimal Journal Performance on DB2 Universal Database for iSeries*, SG246286
- ▶ *WebSphere and .NET coexistence*, SG24-7027
- ▶ *WebSphere MQ Solutions in a Microsoft .NET environment*, SG24-7012

Other publications

This publication is also relevant as an information source:

- ▶ Jeffrey Richter, *Applied Microsoft .NET Framework Programming*, Microsoft Press, 2002, ISBN 0735614229

Online resources

These Web sites and URLs are also relevant as further information sources:

- ▶ DB2 Universal Database for iSeries home page
<http://www.ibm.com/iSeries/db2>
- ▶ DB2 Information Center
<http://publib.boulder.ibm.com/infocenter/db2help/index.jsp>
- ▶ DB2 UDB, DB2 Connect, and DB2 Information Integrator Version 8 product manuals
<http://www.ibm.com/software/data/db2/udb/support/manualsv8.html>

- ▶ DB2 Universal Database: Selected Common SQL Features for Developers of Portable SQL Applications
<http://www.ibm.com/servers/enable/site/db2/db2common.html>
- ▶ IBM Publications Center
<http://www.ibm.com/shop/publications/order>
- ▶ iSeries Information Center
<http://publib.boulder.ibm.com/pubs/html/as400/infocenter.html>
- ▶ iSeries Access for Windows
<http://www.ibm.com/servers/eserver/series/access/windows>
- ▶ Microsoft Library
<http://msdn.microsoft.com/library/>

How to get IBM Redbooks

You can search for, view, or download Redbooks, Redpapers, Hints and Tips, draft publications, and Additional materials, as well as order hardcopy Redbooks or CD-ROMs, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Index

Symbols

- # code point 149
- \$ code point 149
- .NET Framework version 170
- .NET languages 14
- .NET redistributable 13
- ? parameter marker 78
- @ code point 149
- @ parameter marker 76

Numerics

- 5722-XE1 33
- 5722-XW1 34

A

- AcceptChanges 23, 25
- access path 5
- Accessing physical files with multiple members 123
- ADO 149
- ADO.NET 18–19
- ADODB objects 158
- ALIAS 123
- ALTER TABLE 91
- Although 38
- assembly reference 43, 198
- Authentication on the iSeries 49
- Autocommit 119
- AutoComplete 219
- automatic transaction enlistment 118

B

- BeginTransaction 20, 64, 201
- Best practices 146, 223
- Binary large objects (BLOBs) 132, 205
- BIT DATA 90
- BLOB 132, 205

C

- Calling a program 120
- Calling stored procedures 80
- Cancel 70
- CCSID 65535 90, 128
- ChangeDatabase 65
- Character large objects (CLOBs) 132, 204
- CheckConnectionOnOpen 55, 107
- Choosing your Execute method 86
- CL command 120
- CleanupPooledConnections 145, 171
- Clear (DataSet method) 23, 25
- CLOB 132, 204
- Close 20, 65, 201, 203
- CLR 14

- CLS 14–15
- columns in SQL 5
- COM+ 210, 212
- CommandBuilder example 110
- CommandText 20, 67, 201
- CommandTimeout 67
- CommandType 20, 67, 201
- Common Language Runtime 14
- Common Language Runtime features 15
- Common Language Specification 14–15
- Common Type System 15
- communication error 103–104, 106
- communication traces 168
- compression 57, 148
- concurrency violation 138, 150
- Configuration Assistant 181
- Connection 68
- Connection pooling 54, 105, 143, 223
- ConnectionString 19, 48, 200
- ConnectionTimeout 50
- Console Application 43
- CreateCommand 20, 201
- CreateParameter 21, 175, 202
- CTS 15
- culture-specific settings 130
- CurrentCulture 130
- CurrentUICulture 129
- cursor 150
- cwbcotrc utility 168
- cwbmptrc utility 167
- cwbping 36

D

- Data Description Specification (DDS) 5
- data field 5
- Data queues 38
- Data types 37, 87, 196
- DataAdapter 22, 127, 129, 203
 - example 110, 203
- Database 51, 180
- DataCompression 57
- DataLinks 38, 141
- DataReader 21, 127, 202
 - example 107, 202
- DataSet 22
- DataSetName 23
- DataSource 49
- Dates 94
- DB2 UDB for iSeries
 - overview 4
 - programming languages 4
 - sample schema 8
- DB2Command 201
 - Methods 202

- CreateParameter 202
- ExecuteNonQuery 202
- ExecuteReader 202
- ExecuteScalar 202
- Properties 201
 - CommandText 201
 - CommandType 201
- DB2CommandBuilder 207
 - rules for using 207
- DB2Connection 19, 199
 - ConnectionString properties 199
 - Connection Lifetime 199, 223
 - Connection Reset 199, 223
 - CurrentSchema 199
 - Database 199
 - Enlist 199
 - Isolation Level 200
 - Max Pool Size 200, 223
 - Min Pool Size 200, 223
 - Password 199
 - Persist Security Info 200
 - Pooling 200, 223
 - PWD 199
 - Server 199
 - UID 199
 - User ID 199
 - Methods 200
 - BeginTransaction 201
 - Close 201
 - CreateCommand 201
 - Open 201
- DB2DataAdapter 203
 - Methods 204
 - Fill 204
 - Update 204
 - Properties 204
 - DeleteCommand 204
 - InsertCommand 204
 - SelectCommand 204
 - UpdateCommand 204
- DB2DataReader 202
 - Methods 203
 - Close 203
 - Read 203
 - Properties 203
 - FieldCount 203
 - HasRows 203
- DBCLOB 132, 204
- DDS (Data Description Specification) 5
- Decimal separator 100
- DefaultCollection 52–53
- DefaultIsolationLevel 59
- DeleteCommand 22, 204
- delimited identifiers 149
- Deploying your application 146
- DeriveParameters 70
- Dispose 71
- Distributed Transaction Coordinator (DTC) 39, 199
- Distributed transactions 118, 210
- Double-byte character large objects (DBCLOBs) 132,

- 204
- DRDA 39, 178

E

- Exceptions 102, 166
 - DB2Exception 195
 - iDB2CommErrorException 103
 - iDB2SQLException 102
 - Message property 102
 - MessageCode property 103
 - MessageDetails property 102
 - Provider-specific 37
 - SqlState property 103
- Execute 86
- ExecuteNonQuery 21, 71, 202
- ExecuteReader 21, 71, 202
- ExecuteScalar 21, 71, 202
- extended dynamic 38

F

- field 5
- FieldCount 21, 203
- Fill 115, 204

G

- garbage collection 148
- GetXML 23, 25

H

- Handling exceptions 102
- HasRows 21, 203
- HexParserOption 58
- Host setup 40

I

- IASP 51
- IBM DB2 UDB for iSeries .NET provider 33
- IBM Support 169
- IBM.Data.DB2 177
- IBM.Data.DB2.iSeries 33
- IBMDA400 OLE DB provider 38, 152, 159, 171
- IBMDARLA OLE DB provider 38
- IBMDASQL OLE DB provider 38
- iDB2Command 65
 - constructor 66
 - Methods 70
 - Cancel 70
 - CreateParameter 175
 - DeriveParameters 70
 - Dispose 71
 - ExecuteNonQuery, ExecuteReader, and ExecuteScalar 71
 - Prepare 74
 - Properties 67
 - CommandText 67
 - CommandTimeout 67
 - CommandType 67

- Connection 68
 - Parameters 69, 74
 - Transaction 69
 - UpdatedRowSource 69
- iDB2CommandBuilder 139
 - rules for using 139
- iDB2Connection 19, 48
 - ConnectionString properties 48
 - CheckConnectionOnOpen 55
 - ConnectionTimeout 50
 - Database 51
 - DataCompression 57
 - DataSource 49
 - DefaultCollection 52
 - DefaultIsolationLevel 59
 - HexParserOption 58
 - JobName 63
 - LibraryList 52
 - MaximumDecimalPrecision 60
 - MaximumDecimalScale 60
 - MaximumInlineLobSize 58
 - MaximumPoolSize 54
 - MaximumUseCount 55
 - MinimumDivideScale 60
 - MinimumPoolSize 55
 - Naming 51
 - Password 49
 - Pooling 54
 - Provider 62
 - QueryOptionsFileLibrary 59
 - ServerVersion 62
 - SortLanguageId 57
 - SortSequence 56
 - SortTable 57
 - SSL 50
 - State 62
 - Trace 63
 - UserID 49
 - constructor 48
 - Format of the ConnectionString 48
 - Methods 64
 - BeginTransaction 64
 - ChangeDatabase 65
 - CleanupPooledConnections 145
 - Close 65
 - CreateCommand 65
 - Open 65
 - Overriding your ConnectionString 168
- iDB2DataAdapter 127
- iDB2DataReader 71
- iDB2Date 94
- iDB2Decimal 97
- iDB2Exception 169
- iDB2Time 94
- iDB2TimeStamp 94
- IIS 49
- Imports statement 44, 199
- independent auxiliary storage pool (IASP) 51
- index in SQL 5
- inline LOB data 58

- InsertCommand 22, 204
- integrated relational database 4
- Intermediate Language 14
- Internet Information Services (IIS) 49
- iSeries Access for Windows 34
- iSeries Information Center 36, 179
- iSeries isolation level 59
- ISO formats for date, time, and timestamp 94
- Isolation levels 59, 119

J

- JDBC 149
- job description 51
- Joblog 46, 64, 170
- JobName 46, 63, 170

K

- Kerberos 50
- key columns 140
- keyed logical file 5

L

- languages 129
- Large decimal and numeric data 60
- Large objects (LOBs) 58, 128, 132, 204
 - example 132, 204
 - techniques 135
- LibraryList 52
- LOB locators 58
- Local transactions 116, 210
- logical file 5

M

- MaximumDecimalPrecision 60
- MaximumDecimalScale 60
- MaximumInlineLobSize 58
- MaximumPoolSize 54
- MaximumUseCount 55
- method
 - BeginTransaction 20, 64, 201
 - Close 20–21, 65, 201
 - CreateCommand 20, 65, 201
 - ExecuteNonQuery 24
 - Fill 22
 - Open 19, 65, 200–201
 - Read 21
 - Update 22
- Microsoft Intermediate Language 15
- MinimumDivideScale 60
- MinimumPoolSize 55
- MSDN Library Web site 34
- MSIL 15
- MTS 39
- Multiple members of a DDS file, accessing 123
 - using an ALIAS 123
 - using OVRDBF 123
- Multiple results 84

N

named parameter 78
namespace directive 44, 199
Naming convention 51
nullable columns 140
nullable fields 147

O

object

- Command 20
 - DB2Command 201
 - iDB2Command 65
- Connection 19
 - DB2Connection 180
- DataAdapter 22
 - DB2DataAdapter 203
 - iDB2DataAdapter 110
- DataReader 21
 - DB2DataReader 202
 - iDB2DataReader 71
- Parameter 24
 - DB2Parameter 195
 - iDB2Parameter 69

ODBC 149, 172
OLE DB 149
OLE DB properties 151

- Add Statements To SQL Package 151
- Block Fetch 151
- Catalog Library List 151
- ConnectionTimeout 151
- Convert Date Time To Char 151
- Current Catalog 151
- Cursor Sensitivity 151
- Data Compression 151
- Data Source 151
- DBMS Version 151
- Default Collection 151
- Force Translate 151
- Hex Parser Option 151
- Initial Catalog 151
- Job Name 151
- Maximum Decimal Precision 151
- Maximum Decimal Scale 151
- Minimum Divide Scale 151
- Password 151
- Provider 152
- Query Options File Library 152
- Sort Language ID 152
- Sort Sequence 152
- Sort Table Name 152
- SQL Package Library Name 152
- SQL Package Name 152
- SSL 152
- State 152
- Trace 152
- Unusable SQL Package Action 152
- Use SQL Packages 152

OleDbConnection 19
Open 20, 65, 201

optimistic 150
optimistic concurrency 138, 140, 150
outfile, processing 123
Overriding your ConnectionString 168
OVRDBF 123

P

parameter markers 76, 147
Parameters 69, 74, 76
Password 49
PC setup 39
performance 146, 230
pessimistic concurrency 150
physical data 5
physical files 5
Platform Support 13
Pooling 54, 200
Prepare 74, 148
prestart job 35, 179
primary key 140, 147
problem determination 63
program call 38
Provider architecture 34, 178
Provider data types 87, 196
provider independence 171
proxy 220
PTF 170

Q

QAQQINI 59
QCMDEXC 120
Query options file 59
QueryOptionsFileLibrary 59
QUSRWRK subsystem 35, 179
QZDASOINIT 34
QZDASSINIT 35

R

RDB 51
Read 203
ReadXml 23, 25
record field 5
record level access 38
record selection 5
records 5
Redbooks Web site 264

- Contact us xi

relational database

- integration overview 4

Remote command 38
Result data 83, 86
return value parameters 83
rows 5
RowUpdatedEvent 139
run-time authorization identifier 52

S

sample schema 40

- sampledb 40
- Secure Sockets Layer 50
- SelectCommand 22, 204
- Selective Setup 39
- server job 34–35, 178
- ServerVersion 62
- Service Packs 40
- SetAbort 214
- SetComplete 214
- SharedWeight 56
- Sort sequence 132
- SortLanguageId 57
- SortSequence 56
- SortTable 57
- special characters 148
- spool file, processing 124
- SQL 5
 - view defined 5
- SQL (Structured Query Language) 5
- SQL columns 5
- SQL errors 102
- SQL index 5
- SQL naming 51–52
- SQL packages 38
- SQL table 5
- SQL view 5
- SQLCURRULE 58
- SqlState 169
- SSL 50
- State 62
- Stored procedures 79, 147
- STRHOSTSVR 35
- Structured Query Language (SQL) 5
- Supported features 36, 194
- supported host versions 40, 178
- System naming 52

T

- table
 - in SQL 5
 - rows 5
- Technical Reference 39, 41, 198
- Time 128
- TIME and TIMESTAMP special value 24
 - 00
 - 00 96
- Time and Timestamp special values 128
- timeout 67
- Times 94
- Timestamp 128
- Timestamps 94
- Trace property 63
- Tracing the provider 167
- Transaction 69
- transaction mode 59
- Transactions 116
 - and stored procedures 119
 - distributed 118, 210
 - Isolation levels 119
 - local 116

- try/catch 166

U

- UDTs 38
- unique key 147
- Unique keys 140
- UniqueWeight 56
- unnamed parameter 78
- unqualified object names 51–52
- Unsupported features 38, 196
- Updatable recordset 159
- Update 204
- UpdateCommand 22, 204
- UpdatedRowSource 69
- Updating DataSets 136
- User defined types (UDTs) 38
- UserID 49
- using directive 44, 199
- Using large objects (LOBs) 132

V

- view in SQL 5
- view of physical data 5
- Visual Studio .NET Add-Ins 182
 - Data Controls 186
 - DB2 Database Project type 184
 - IBM Explorer 186
 - Registering 183
 - Toolbar 184
 - Unregistering 183

W

- WriteXml 23, 25
- WRKJOB 47

X

- XML 23, 27



Integrating DB2 Universal Database for iSeries with Microsoft ADO .NET

(0.5" spine)

0.475" <-> 0.873"

250 <-> 459 pages



Redbooks

Integrating DB2 Universal Database for iSeries with Microsoft ADO .NET

Discover the power of ADO .NET Data Providers for the iSeries

Learn best practices, performance tuning, and migrating from OLE DB

Master iSeries .NET programming

Customers have been using the IBM DB2 UDB for iSeries for many years with data access technologies such as ODBC and OLE DB. The newest data access technology from Microsoft is called ADO.NET. Applications that use ADO.NET with the iSeries can work with several different .NET providers:

- ▶ The IBM.Data.DB2.iSeries provider, a .NET-managed provider new to iSeries Access for Windows in V5R3
- ▶ The IBM.Data.DB2 provider, a .NET provider that works with all IBM @server platforms in conjunction with DB2 Connect
- ▶ The Microsoft System.Data.OleDb provider, as a bridge to one of the OLE DB providers included with iSeries Access for Windows (IBMDA400, IBMDASQL, and IBMDARLA)
- ▶ The Microsoft System.Data.Odbc provider, as a bridge to the ODBC driver included with iSeries Access for Windows

This IBM Redbook shows customers how to use ADO.NET effectively to harness the power of DB2 UDB for iSeries, showing examples, best practices, pitfalls, and comparisons between the different ADO.NET data providers.

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks