CHAPTER 6

Disconnected Data via ADO.NET and DataSets

Introduction

During the beta for Windows 2000, Microsoft released a very exciting part of COM+ that would allow for the manipulation of disconnected data in memory. This technology—called IMDB (in-memory database)—promised to solve a host of problems associated with distributed database applications, as it meant that an application could connect to a data source, retrieve into memory the necessary data, disconnect from the data source, and then read from and manipulate the in-memory representation of the data without incurring constant round-trips to the data store or keeping the connection alive for long periods of time, thereby locking crucial data. Once finished with its work, the application could then connect the IMDB back to its data store and synchronize any changes that were made. Unfortunately, just as Release Candidate 2 for the Windows 2000 operating system was being released, Microsoft announced that for various reasons IMDB just wasn't ready for prime time and would not ultimately be a part of the Windows 2000 final release.

While many of us were very disappointed by this decision, what we could not have known at the time was that IMDB was taking a slightly different—and more powerful—form and being included into something then called NGWS (Next Generation Windows Services)—what we now know as .NET. This .NET incarnation of an in-memory, disconnected data model is only one part of the overall ADO.NET data access layer. However, as MFC developers, we have plenty of data access technologies to pick from (MFC database classes, ODBC, OLEDB, ADO, DAO, etc.). Therefore,

this chapter will focus on the aspects of ADO.NET that we do not have—specifically the various classes that support the original concepts of IMDB.

The chapter begins with a section illustrating the basic terminology and classes specific to ADO.NET and disconnected data as well as the basics of constructing those objects and connecting to a data source. From there, you'll begin a section illustrating how to perform basic database operations (creating, reading, updating, and deleting of records). Included in that section are tips on handling the common issues of dealing with disconnected data and auto-increment primary keys, and how to specify primary key information for untyped datasets. The section ends with a fully functional MFC maintenance application using the ADO.NET against a sample database. At this point, you'll be fairly comfortable with the basics of reading and writing data to a disconnected data table. I'll then illustrate several techniques for reading and writing data in batch in order to keep round trips to the data server to a minimum and dramatically speed up insert operations. One of the key benefits of an IMDB is its support of the ability to search, sort, and filter data once it's been retrieved. Therefore, the next section illustrates how to accomplish these tasks as well as how to create multiple filtered views on a single dataset. Finally, the chapter ends with a section on generating strongly typed datasets, using them in an MFC application, and the pros and cons of using untyped vs. typed datasets.

ADO.NET Basics

As programmers, many of us like to jump right in and start using a technology with so much promise. However, logic dictates that we take a minute to familiarize ourselves with the terms and classes that implement this technology. Therefore, this section is meant as a primer for the rest of the chapter. I'll begin by quickly going over the terminology and main classes associated with disconnected data and then introduce a few tasks and concepts that will be used or referenced throughout the chapter.

ADO.NET Terminology and Main Classes

The first new term you'll hear quite often regarding ADO.NET is that of a *managed provider*. This is simply the .NET equivalent of terminology that was originally introduced with OLEDB (and later used by its COM

interface, ADO). In OLEDB, code that provides a generic interface to data is referred to as a *provider.* Therefore, since code written to run on top of the CLR is called "managed," we are given yet another new database term to remember. As of the time of this writing, the .NET Framework defines five managed providers:

- **OLEDB:** Supports data stores that have an OLEDB provider.
- **ODBC:** Supports data stores that have an ODBC driver.
- **Oracle:** A set of classes optimized for the Oracle database product.
- **SQL CE:** A .NET Compact Framework managed provider that supports Microsoft SQL Server CE.
- **SQL Server:** A set of classes that are optimized to support the Microsoft SQL Server database product.

While we're on the topic, I'll also be using the familiar terms *data source* and *data consumer.* (*Data source* is the generic name for the data being provided for consumption by the consumer.) Obviously, the consumer is any code that retrieves, stores, and manipulates data represented by the managed provider.

Like many other frameworks that you've seen throughout this book, ADO.NET is comprised of many classes. However, this chapter will focus on the following classes:

- **`Connection`**: Functions much like the ADO object of the same name and represents a connection to a data source.
- **`Command`**: Another holdover from ADO, the `Command` object represents a query or a command that is to be executed by a data source.
- **`CommandBuilder`**: Used to automatically generate the insertion, update, and delete commands for the data adapter object based on the select command. It is also used to provide optimistic concurrency for disconnected `DataSet` objects.
- **`DataSet`**: One of the key elements with ADO.NET is the `DataSet`. A little too involved to be defined with a single sentence, the `DataSet` represents an in-memory model of disconnected data and has built-in support for XML serialization. That latter capability is covered in Chapter 8, "Combining ADO.NET and XML."
- **`DbDataAdapter`**: The abstract base class for all data store–specific classes such as `SqlDataAdapter`, `OracleDataAdapter`, `OleDbDataAdapter`, and so on.
- **`DataAdapter`**: The base class for the `DbDataAdapter` class.

- **Data adapter**: Not really a class, but a generic designation for one of the `DbDataAdapter`-derived classes.
- **`DataView`**: This class is most easily defined to MFC developers as the data equivalent of a `CView` class for data. For example, in a standard MFC document/view class you can build multiple views that are built on—but work with different parts of—the same data. Likewise, multiple `DataView` objects represent different views on the same `DataSet`.
- **`XmlDataDocument`**: Enables you to treat `DataSet` data as XML data in order to support things like `XPath` search expressions, XSL (eXtensible Stylesheet Language) transformations, and so on.

Now that you've been introduced to the terms, it's easier to define a managed provider as a group of classes that interface to the generic `DataSet` class to abstract you from the specifics of the data you are reading or modifying. For example, the `System::Data::SqlClient` namespace defines about 15 classes and several delegates that are optimized for use with the SQL Server database product. Among these classes are derived types of the base classes I mentioned in the previous list: `SqlConnection`, `SqlDataAdapter`, `SqlCommand`, and `SqlCommandBuilder`.

Let's now look at the `DataSet` class a bit more closely. The `DataSet` class is a collection of data structures (other classes) that are used to model relational data. The following list details the main classes that comprise either the `DataSet` class or one of its member classes:

- **`DataTable`**: If you're familiar with ADO, then at first glance you might be tempted to think of a `DataSet` class as being comparable to souped-up ADO `Recordset` objects. However, datasets are so encompassing that there is no equivalent in ADO for them. The `DataTable` class, on the other hand, is a more true ADO.NET equivalent of the ADO `Recordset` object, as it encapsulates a two-dimensional array (rectangle) of data organized into columns and rows.
- **`DataColumn`**: Within the `DataTable` class are a collection of `DataColumn` definitions. As the `DataRow` class (described next) defines actual data, the `DataColumn` class defines the data store column definitions. Example members of this class are `ColumnName` and `DefaultValue` as well as Boolean properties such as `AllowDBNull`, `AutoIncrement` and `ReadOnly`.

- **DataRow**: The `DataRow` class encapsulates the data for a given `DataTable` object in addition to defining many members that support the disconnected capabilities of the `DataSet`/`DataTable`. These members include support for tracking the current and original values of each column, the current state of the row (a `DataRowState` enumeration with such values as `Added`, `Deleted`, `Detached`, `Modified`, and `Unchanged`) and a connection to the parent table to support `DataRelation` via the `GetParentRows` and `GetChildRows` methods
- **DataRelation**: `DataRelation` objects are used to define how multiple `DataTables` are associated. For example, it is quite common to use this feature when dealing with tables that have a parent/child relationship, such as order header and order detail tables. Using this feature, you can more easily navigate the related data of these two tables. This class is covered in more detail in the next chapter.
- **Constraint**: Each `DataTable` defines a collection of constraints that specify rules for maintaining data integrity. For example, when you delete a value that is used in one or more related tables, a `ForeignKeyConstraint` determines whether the values in the related tables are also deleted, set to null values, set to default values, or whether no action occurs.

Constructing and Filling `DataSet` Objects

Now that that you've been introduced to the main ADO.NET classes that will be used throughout this chapter, let's take a look at a code snippet that illustrates how to connect to and retrieve data from a data source. After the code snippet, I'll provide a walkthrough of the various classes that are being used here as well as a lot of not-so-obvious tasks that are being performed for us in order to facilitate a disconnected dataset.

```
SqlConnection* conn =
  new SqlConnection(S"Server=fantine;"
                    S"Database=Northwind;"
                    S"Integrated Security=true;");
SqlDataAdapter* adapter =
  new SqlDataAdapter(S"SELECT * FROM Employees", conn);
SqlCommandBuilder* cmd = new SqlCommandBuilder(adapter);
conn->Open();
```

```
DataSet* dataset = new DataSet();
  adapter->Fill(dataset, S"AllEmployees");
conn->Close(); // No longer needed
DataTableCollection* tables = dataset->Tables;
employeesTable = tables->Item[S"AllEmployees"];
// ... Use employees table as needed.
```

While this code looks pretty straightforward, there's much more going on here than meets the eye.

- The first thing the code snippet does is to connect to the SQL Server sample database, Northwind, using the `SqlConnection` class.

Specifying Connection Strings for Different Database Products

For this chapter, I chose to use the SQL Server database as it's the most commonly used database among Visual C++/MFC professionals. In addition, while much of the code that you'll see in this chapter can easily be massaged to work with any managed provider, the initialization of the `Connection` object is data source–specific. Therefore, if you are using another product, such as Oracle or Microsoft Access, or want to use the OLEDB or ODBC interfaces to these or other databases, the http://www.connectionstrings.com Web site is an invaluable resource, as it contains connection strings for virtually every data store.

- Once that is done, the code uses a `DbDataAdapter`-derived class (`SqlDataAdapter`) designed specifically for SQL Server access. As mentioned in the previous section, the data adapter is what connects a dataset to the underlying data store. However, what's really interesting here is that while I'm passing a "select" value to the `SqlDataAdapter` class' constructor, the various data adapter classes define four distinct commands (in the form of `SqlCommand` classes): `SelectCommand`, `InsertCommand`, `UpdateCommand`, and `DeleteCommand`. (From here on, the latter three commands will be referred to en masse as *action commands*.) One extremely important note to make here is that *the data adapter does not automatically generate commands to reconcile changes made to a dataset based on the `select` statement used to construct the adapter.* You must either set these commands yourself or use a command builder class, which segues nicely into the next items of interest from the code snippet.

- Once the data adapter has been constructed with the desired select command, an `SqlCommandBuilder` object is instantiated and associated with the data adapter. The command builder automatically generates the appropriate action commands (complete with the underlying SQL code, ADO.NET `Command` objects, and their associated `Parameters` collections) based on the adapter's `SelectCommand`.
- Next, the connection is opened. One thing to note here is that the data adapter is designed to minimize the time a connection stays open. As you see more code in this chapter, take note that the data adapter's associated connection is never explicitly opened or closed. Instead the adapter knows when it needs to connect and disconnect. For example, when calling the data adapter object's `Update` method in order to commit changes to the dataset, the data adapter will automatically use an already open connection to the data store or make the necessary connection and automatically disconnect when finished.
- After that, we're finally down to the `DataSet` object itself. To construct and fill the `dataset`, you can simply use the `DataSet` class's default constructor and then call the data adapter object's `Fill` method, passing the constructed `DataSet` object as the first parameter. The `Fill` method retrieves data from the underlying data store based on the data adapter's `SelectCommand` value. (In this example, that value was set when the `SqlDataAdapter` was constructed.) You'll also notice that I specified a literal value of `"AllEmployees"` for the second parameter to the `Fill` method. This value specifies the name that I wish to give the `DataTable` that will be constructed with the returned data. If I had not named the dataset's data table, it would have been named "Table" automatically. (When more than one data table are generated and not specifically named, they are assigned the names Table1, Table2, and so on.)

Creating Multiple `DataTables` in a `DataSet`

While most of the chapter's code snippets and demo applications will only read and modify a single table, there might be times when you'll want a `DataSet` to contain multiple `DataTable` objects. The section entitled "Creating Multiple `DataTable`s in a `DataSet`" will illustrate how to do this both by using multiple data adapters and also by combining multiple `SELECT` statements in a single data adapter in order to reduce round trips to the server.

- At this point, the requested data is in the `DataRow` members of the `DataTable` members of the `DataSet` object. Therefore, the code can safely disconnect from the data source and continue working until it wants to commit any changes made to the data!
- The last thing I'll illustrate here before moving on to the next code snippet is how to retrieve the desired `DataTable` objects from the `DataSet` object. As you can see from the code, the `DataSet` class has a public property called `Tables` that is simply a collection of `DataTable` objects. As with accessing any other .NET collection with Managed Extensions, you can use one of two overloaded `Item` indexers—one accepts the relative index and the other the named entity. Therefore, as the data adapter in this code snippet only constructed a single `DataTable` object that was named `"All Employees"` in the `Fill` method, it can be retrieved either by name or by passing its index value of 0.

Different Ways to Construct Datasets

There are three distinct methods to constructing and filling datasets. One way—used in this chapter—is from a data adapter (which is typically associated with a database). You can also construct a dataset programmatically from any data your application has access to, either read from another source or generated within the application. This technique—while not overly difficult—is not used very often and is beyond the scope of this chapter. Finally, you can also construct a `DataSet` object from an XML document in situations where you wish to treat XML data as you would any other data format. The topic of mixing ADO.NET and XML is covered in Chapter 8.

Untyped vs. Typed Datasets

There are two basic ways to use the `DataSet` objects: *untyped* and *typed.* When using untyped datasets, you use the base BCL-provided `DataSet` objects and pass the relevant information that specifies which table, column, row, and so on that you're working with. For example, let's say that you're working with a row of data (represented by a `DataRow` object) for a table that contains a column named `FirstName`. For each row, you could access and modify the `FirstName` column as follows:

```
// row is a DataRow object
// Retrieve value
```

```
String firstName = row->Item[S"FirstName"]->ToString();
// Set value row->Item[S"FirstName"] = S"Krista";
```

The `DataRow`—needing to be a generic interface for all data—provides methods for reading and updating column values, respectively, where you're responsible for specifying the column name and—if updating—an `Object` representing the value. This generic approach, which makes you responsible for the specifics, is used throughout all the `DataSet` classes. Therefore, the main drawback to untyped datasets is that the code is not type-safe. In other words, mistakes made in your code, such as misspelling the column name or passing an incompatible data type, will only be realized at runtime.

Typed datasets, on the other hand, are classes that are generated from a specified data store. It's important to realize that these classes are still directly derived from the ADO.NET base classes. However, they contain members specific to the data store schema and, as such, allow for compile-time error checking. To continue our `Employees` table example, a typed `DataSet` would include a `DataRow`-derived class called `EmployeesRow`. This class would then define members for each column in the `Employees` table, as shown in the following excerpt.

```
public: EmployeesDataSet::EmployeesRow*  AddEmployeesRow
(
  System::String*  LastName,
  System::String*  FirstName,
  System::DateTime HireDate,
  System::Byte Photo[],
  System::String*  Notes,
  System::Int32 ReportsTo
);
```

Using the typed dataset, our read and update code becomes the following:

```
// row is an EmployeesRow object
// Retrieve value
String firstName = row->FirstName;
// Set value row->FirstName = S"Krista";
```

As you can see, the main benefits to typed datasets are better readability and compile-time type checking—as each column is a class member

that is associated with its correct type within the class. To draw a parallel between typed datasets and our MFC world, you could say that typed datasets are analogous to using the MFC ODBC Consumer Wizard to generate a `CRecordSet` class. The main difference is that while the various ADO.NET classes can be bound to .NET Windows Forms controls, they were designed for a managed world; thus, there's nothing akin to RFX that will automatically bind the data to our MFC dialogs/views and controls. That we have to do manually.

I'll get into more of the advantages and disadvantages of using typed datasets in the section entitled "Working with Typed Datasets." However, I at least wanted you to know at this point that they both exist and to understand the main differences between them. Also note that while typed datasets have some obvious advantages, this chapter will use mostly untyped datasets for the following reasons:

1. Untyped datasets allow you to see more easily what is really going on in code snippets as the client code explicitly states table and column names, store-procedure parameter names, and so on, as opposed to the actual database entity names being hidden in a class.
2. Untyped datasets allow for shorter, more focused code snippets and demos. Otherwise, each demo would require extra steps to create the typed datasets and then would require a lot of cross referencing between the main code and the typed `DataSet` class code.

Basic Database Operations with ADO.NET

Whether you're working with a connected or disconnected data store, the majority of database operations involve **NURD** work—**N**ew, **U**pdate, **R**ead, **D**elete. However, as this section will illustrate, many of the sometimes very tedious database operations are made much easier with the help of the various ADO.NET classes.

Quick Note on This Section's Examples

This section's code snippets are all freestanding functions that can be plugged directly into your own test applications. They make the sole assumption that the `DataSet`, `DataAdapter`, `DataTable`, and `CommandBuilder` objects have all been properly

constructed. For example, in an SDI application you might declare each of these as member variables of the view and instantiate them (as illustrated in the section entitled "Constructing and Filling DataSet Objects") in the view's `OnInitialUpdate` function. This way, you can follow along, trying the various code snippets without having to see the same connection code repeated over and over in each code example.

Reading Data

The first "real" task that many people new to a given database access layer want to explore is that of reading data. In the previous sections, you learned how to connect to a data store and construct a `DataSet` object that in turn contains a collection of `Table` objects. That is where we'll pick up here—the objects within the `Table` object.

The `Table` class contains two integral collections that you'll use most often: one for holding all columns (`DataColumnCollection`) and one for holding all returned rows from a query or command (`DataRowCollection`). Within these collections are held the `DataColumn` and `DataRow` objects, respectively. Figure 6–1 shows the relationship between these classes.

Let's start with the `DataRowCollection` and `DataRow` classes. The `DataRowCollection` is simply the collection of `DataRow` objects returned from the query or command executed by the adapter. This class is used to

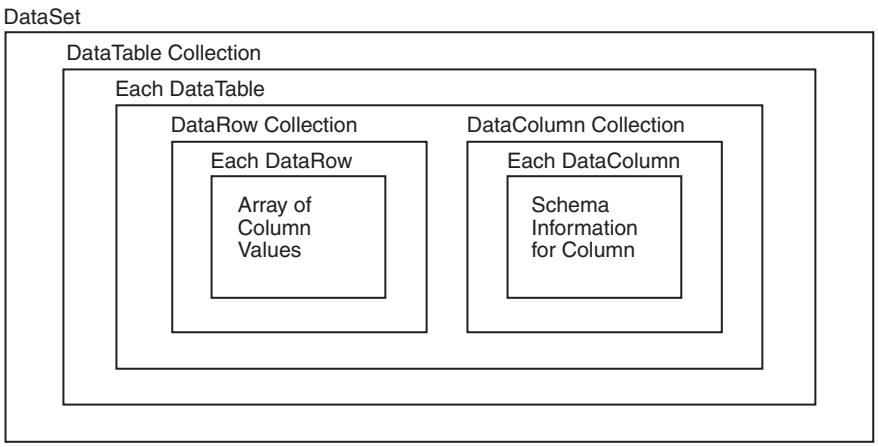

Figure 6–1 Basic relationship between `DataSet`, `DataTable`, `DataRow Collection`, `DataRow`, `DataColumnCollection`, and `DataColumn` classes

perform operations on the entire set of rows, such as inserting new rows, deleting rows, and searching for rows. Each of these tasks will be covered shortly. For now, we'll just be enumerating this collection in order to get at the row objects. To see how these two classes are used to read elements from a `DataSet`, take a look at the following function (`ListAllEmployees`) that connects to the sample SQL Server database Northwind, enumerates its Employees table, and retrieves the `FirstName` and `LastName` columns for each row:

```
void ListAllEmployees()
{
#pragma push_macro("new")
#undef new
  try
  {
    // Assumes employeesTable has already been filled by an adapter
    DataRowCollection* rows = employeesTable->Rows;
    DataRow* row;
    String* id;
    String* firstName;
    String* lastName;

    for (int i = 0; i < rows->Count; i++)
    {
      // Get DataRow object
      row = rows->Item[i];

      id = row->Item[S"EmployeeID"]->ToString();
      firstName = row->Item[S"FirstName"]->ToString();
      lastName = row->Item[S"LastName"]->ToString();
    }
  }
  catch(Exception* e)
  {
    MessageBox::Show(String::Format(S"Exception : {0}", e->Message));
  }
#pragma pop_macro("new")
}
```

As you can see, once the `ListAllEmployees` function retrieves the `DataRowsCollection` object from the Employees `DataTable` object (via the `Rows` method), it then employs a `for` loop to enumerate the collection.

Within the loop, the `DataRowCollection::Item` property is called in order to retrieve each `DataRow` object from the collection. Once the `DataRow` object has been secured, the `DataRow::Item` property is used to retrieve the desired column's data (by column name). As you can see, once the higher-level objects (such as `DataSet` and the data adapter) have been constructed, the database code resembles any other—except that here you have the power of disconnected data.

Now let's look at one way the `DataColumnCollection` and `DataColumn` classes can help in reading data. As the `DataColumnCollection` object contains an array of every `DataColumn` object for a given `DataTable`, you can easily determine column-level information when needed—such as when coding an agnostic client—or one that has no knowledge of the data store's schema. By interrogating the `DataColumn` object, you can determine many important aspects of the column's definition, including column name, data type, whether or not the column is read-only, has a default value, and so on.

Take a look now at the following method, which illustrates how the `DataColumnCollection` and `DataColumn` classes can be used to dynamically determine column information and retrieve all data from a given table (the Employees table, in this case).

```
void DumpEmployeeTable()
{
  try
  {
    // Assumes employeesTable has already been filled by an adapter
    // Get all column names and column types...
    String* columnName;
    String* columnType;
    for (int i = 0; i < employeesTable->Columns->Count; i++)

    {
      // Get column name
      columnName = employeesTable->Columns->Item[i]->ColumnName;
      // Get column data type
      columnType =
        employeesTable->Columns->Item[i]->DataType->ToString();
      // Display column information
    }

    // Get all rows and within each row, all column data
    DataRowCollection* rows = employeesTable->Rows;
```

```
    DataRow* row;
    for (int i = 0; i < rows->Count; i++)
    {
      row = rows->Item[i];
      for (int j = 0; j < row->ItemArray->Count; j++)
      {
        String* value = row->Item[j]->ToString();
        // Display column data for current row
      }
    }
  }
  catch(Exception* e)
  {
    MessageBox::Show(String::Format(S"Exception : {0}", e->Message));
  }
}
```

The `DumpEmployeeTable` function begins by enumerating the `employeeTable` object's `DataColumnCollection` with a `for` loop. Within that loop, each column is retrieved with a call to the `DataColumnCollection::Item` property. The `ColumnName` and `DataType` properties are then used to retrieve those values.

Once the column information has been acquired, the function loops through the `employeeTable` object's `DataRowCollection` in similar fashion to what you saw in the `ListAllEmployees` function. The main difference here being that instead of hard-coding the desired columns, the loop contains an inner loop to enumerate each row's columns. Within the inner loop, the overloaded `DataRow::Item` property that takes an array index value is used.

As you can see, a few trivial tweaks and this function could be modified to dump both the complete schema information and data of any table.

Inserting and Updating Rows

Having seen a bit of the `DataRowCollection` class, you might imagine that adding rows to a `DataTable` is easy, and you would be correct. There are two distinct means of inserting new rows into a `DataTable`, each facilitated by an overload of the `DataRowCollection::Add` method:

```
virtual DataRow* Add(Object* valueArray[]);
void Add(DataRow* newRow);
```

The first overload takes an `Object` array of values and returns a `DataRow` object representing the new data row. Here's an example of using this method. (Notice that in the code I'm not defining the first value of the array; I'll explain why shortly.)

```
try
{
  // Assumes employeesTable DataTable has already been constructed
  // Allocate enough elements for all the columns
  Object* values[] = new Object*[employeesTable->Columns->Count];
  // Populate the array
  // Intentionally skipping first element of array
  values[1] = S"Tom";
  values[2] = S"Archer";
  // ...

  // Add the row to the DataTable object
  DataRow* newRow = employeesTable->Rows->Add(values);
  // Must call DataAdapter::Update when ready to commit changes
  // to disconnected data source
  // adapter->Update(dataset, S"AllEmployees");
  //

  MessageBox::Show(S"Record added successfully");
}
catch(Exception* e)
{
  MessageBox::Show(String::Format(S"Exception : {0}", e->Message));
}
```

As you can see, this code snippet first allocates an array of `Object` types using the Managed C++ syntax for allocating an array of reference types, as discussed in Chapter 1. The `DataTable::Count` property is used to ensure that the proper number of elements is allocated (although in this example I only output a couple of values for example purposes). From there, the code populates the array and calls the `Add` method, with the new `DataRow` object being returned.

However, there a couple of key issues to cover here. First, note the comment regarding the data adapter object's `Update` method. Calling the `Update` method will obviously cause the data adapter to connect to the underlying data store in order to reconcile changes in the adapter's specified `DataTable`. Therefore, where you place this logic will be

application-specific. As an example, let's say you have a distributed application where you want to keep connections to the remote data store to a minimum. Instead of calling `Update` on every data change, you could place a UI element on the application (such a Commit Changes menu item) that calls the `Update` method and causes all updates, inserts, and deletes to be reconciled en masse.

The second issue to take note of is that of skipping the first element of the array. Typically as C++ programmers we would cringe to see someone allocate an array, not initialize the first element, and then pass that array to another function for further processing. However, in this case I know that the first element will not be used in the insert of the new record. This can be verified by inspecting the `CommandBuilder` object's `InsertCommand` (via the `GetInsertCommand` method). As shown here, note that the `EmployeeID` column is not specified in the SQL `INSERT` command.

```
"INSERT INTO Employees( LastName , FirstName , Title ,
TitleOfCourtesy , BirthDate , HireDate , Address , City , Region ,
PostalCode , Country , HomePhone , Extension , Photo , Notes ,
ReportsTo , PhotoPath ) VALUES ( @p1 , @p2 , @p3 , @p4 , @p5 , @p6 ,
@p7 , @p8 , @p9 , @p10 , @p11 , @p12 , @p13 , @p14 , @p15 , @p16 ,
@p17)"
```

However, that does bring up an interesting issue that often plagues users of tables with a primary key column that is defined as auto-increment. Specifically, the problem is determining how to insert a new record and then retrieve its auto-incremented primary key. For example, you'd want to do this if you had related tables where the primary key for one table was to be used as part of the key for another table. Also, you might need the primary key in your code in order to programmatically keep track of the records, or you might even need to return that value to the user. I'll cover a common technique for handling this situation in the section entitled "Disconnected Data and Auto-Increment Primary Keys."

Now, let's look at a code snippet that inserts new records into a `DataTable` object by first creating and populating a `DataRow` object. As a `DataRow` can be populated using either the `DataRow:: Item` property (one element at a time) or by using the `ItemArray` property (where an array of `Object` types can be specified), I'll illustrate both techniques.

```
try
{
  // Assumes employeesTable DataTable has already been constructed
```

```
  // Construct a DataRow object using an Object array
  DataRow* newRow1 = employeesTable->NewRow();
  Object* values[] = new Object*[employeesTable->Columns->Count];
  values[1] = S"Tom";
  values[2] = S"Archer";
  newRow1->ItemArray = values;
  employeesTable->Rows->Add(newRow1);
  // Construct a DataRow object one element at a time
  DataRow* newRow2 = employeesTable->NewRow();
  newRow2->Item[S"FirstName"] = S"Nishant";
  newRow2->Item[S"LastName"] = S"Sivakumar";
  employeesTable->Rows->Add(newRow2);
  MessageBox::Show(S"Records added successfully");
}
catch(Exception* e)
{
  MessageBox::Show(String::Format(S"Exception : {0}", e->Message));
}
```

With three row-insertion techniques to choose from, it's logical at this point to question which one is best. My personal recommendation is to use the `NewRow` method and then fill the returned `DataRow` object via the overloaded `Item` that allows you to specify the column name. This way, you eliminate common errors associated with assigning a value to an incorrect column. In addition, explicitly naming the column to which a value is being assigned results in code that is more readable—and therefore, more maintainable.

Deleting and Removing Rows

There are actually three distinct ways to delete (or remove) a row depending upon your needs. Here are the three methods and a brief explanation of their differences, specifically with regard to when you would use one over the other:

- `DataRowCollection::Remove`
- `DataRowCollection::RemoveAt`
- `DataRow::Delete`

First off, the only difference between the `Remove` and `RemoveAt` methods is that the former takes a `DataRow` object as its only parameter, and the latter takes an index value, or ordinal, as its sole parameter. I would

encourage you to be very careful when using the `RemoveAt` method, since adding rows to and removing rows from the `DataRowCollection` will have an impact on the ordinal value of the current rows in the collection. Therefore, unless you are very sure of the index of the row you wish to delete, I recommend using one of the search methods (see "Searching, Sorting, and Filtering Data") to acquire the desired `DataRow` object and then passing that object to the `Remove` method.

```
// Assumes employeesTable DataTable has already been constructed
// and that you have acquired the desired row to delete
employeesTable->Rows->Remove(row);
// Assumes employeesTable DataTable has already been constructed
// and removes the first row in the table's row collection
employeesTable->Rows->RemoveAt(0);
```

One important fact to keep in mind here is that the `Remove`/`RemoveAt` methods do not actually delete the row from the underlying data store upon calling the data adapter object's `Update`. Instead, the `Remove`/`RemoveAt` methods simply set the `DataRow::State` to `DataRowState::Detached` and remove the row from the row collection. In other words, calling the data adapter `Update` method to delete rows only works for rows that are still in the collection and marked as `DataRowState::Deleted`. This is accomplished via the `DataRow::Delete` method:

```
// Assumes employeesTable DataTable has already been constructed
// and that you have acquired the desired row to delete
row->Delete(); // delete row
```

It's worth reiterating that since the data is disconnected, the deletion of the row from the data store will not occur until you call data adapter's `Update`. Also, if you call the `Delete` method and then query either the `DataRowCollection::Count` property or enumerate the `DataRowCollection` object, it will appear as though the record has not been deleted. This is because when you directly access the `DataRowCollection`, you see all records in that collection regardless of their `State` value (`Added`, `Changed`, `Deleted`, etc.).

So how do we know if a record has been deleted so that it is not included in the enumeration of a row collection? This is done via the `DataView` object. I'll discuss the usage of `DataView` objects in more detail in the section entitled "Sorting, Searching, and Filtering Data with `DataView` Objects"). For now, however, I'll describe how the `DataView`

object allows you to filter data rows based on their `DataRowState` property. In fact, every `DataSet` has a "built-in" default data view that's accessible via the `DataTable::DefaultView` property. The default view's `RowState Filter` is set to `DataViewRowState::CurrentRows` so that the view includes unchanged, new, and modified rows, but excludes deleted rows. You can see this in the following code snippet and comments:

```
// Assumes employeesTable DataTable has already been constructed
// and that you have acquired the desired row to delete
row->Delete(); // delete row
// Will be the same as the number before the delete because
// no view is being applied
int dataRowCount = employeesTable->Rows->Count;
// Will be the one less than the count before the delete
// as the default view is set to DataViewRowState::CurrentRows
int defaultViewCount = employeesTable->DefaultView->Count;
```

So the question of which approach you should use really comes down to what capabilities your application will provide to the user. For example, if the application will allow the user to undo delete operations, then you should use the `Delete` method to delete rows and *not* use the `Remove`/`RemoveAt` method, as it would permanently remove the row from the collection. You could then use two `DataView` objects to view your data—one using a filter that ignores rows marked as deleted, and one using a filter that only includes deleted rows. I'll go into this in more detail in the section entitled "Sorting, Searching, and Filtering Data with `DataView` Objects."

If, on the other hand, you have no need to allow for the revocation of delete operations, then you can either remove or delete the row(s). Just make sure that you understand that removing a row from the rows collection means that no matter what filter you set for your view, you will not be able to view the row. Finally, if you are going to delete *and* remove the row, you must call the `Delete` method first.

Disconnected Data and Auto-Increment Primary Keys

There might be times when you work with a table that contains a primary key column defined as *auto-increment* (also known as *auto-number*, or *IDENTITY*). This can be especially problematic for disconnected data scenarios where the application needs to determine the auto-increment value once the record has been inserted into the data store. Here are two examples of such scenarios:

- An application where the record's primary key is used to programmatically keep track of records. For example, a maintenance application might display records in a list and need to retrieve the user-selected record from the data row collection. A newly created record could be inserted into the list with a special value indicating that it was a new record. However, once the user chooses to update the data store, the application-supplied primary key would be incorrect, and the application would need a means of retrieving the new row's actual auto-incremented primary key from the data store.
- An application that uses multiple tables that have a parent-child relationship—such as order header and order detail tables. This is especially problematic if the parent table has an auto-incremented primary key that must be used in inserting rows into the child table.

In both cases, the technique used to determine the auto-incremented value of a newly inserted row is the same. In fact, the first scenario is the subject of the next section's demo application. The second scenario is a bit more complex and brings into play a special ADO.NET class called a `DataRelation`, which enables an application to define a programmatic relation between two tables—a parent and child—such that the data can be navigated in a much more intuitive manner. I'll get into using the `DataRelation` class in the next chapter.

To understand how you can retrieve the auto-incremented primary key value from the data store, it's important to understand the complete order of execution that occurs when a row is updated against a data store.

- The `DataRow` object's values are moved to the adapter's parameter members. These members govern the SQL statement executed against the data store.
- **The data adapter's `OnRowUpdating` method fires the `RowUpdating` event.**
- The command is executed against the data store.
- The `DataRow` object is updated accordingly if the `FirstReturnedRecord` property is set or if any output parameters are specified.
- **The data adapter's `OnRowUpdated` method fires the `RowUpdated` event.**
- The `DataRow::AcceptChanges` method is called to commit the changes to the row.

As you can see, I've made two lines of interest here bold, where an event is fired both before and after the command is executed against the data store. As we're looking for a value returned as a result of executing a command, we're only interested in the second event; therefore, we need only handle the data adapter's `RowUpdated` event.

In .NET you handle an event by registering, for that event, a method that conforms to a *delegate.* A delegate is a signature for a method and is used in cases where one piece of code (a server) needs to define the signature for a method that will be passed to it from another piece of code (a client). Typical uses of delegates include callback scenarios or asynchronous event-handling situations. As a result, the event-handling method must be a managed method.

A common way of writing event-handling methods in mixed-mode applications is to define a managed class that specifies (and implements) the method that's called when the event is fired. Here are the steps involved in defining such a class to handle the `RowUpdated` event and retrieve the auto-incremented primary key for a newly inserted row.

1. Create a managed class that will register and implement the delegates that will handle the desired data adapter events.

```
__gc class SqlEventHandler
{
};
```

2. Implement a class constructor that takes a data adapter as a parameter and register (with that adapter) the method to be called for the desired events. Since we're handling the `RowUpdated` event, that means the code needs to call the data adapter's `add_RowUpdated` method and pass it an `SqlRowUpdatedEventHandler` object. The `SqlRowUpdatedEventHandler` constructor takes an instance of an object that will handle the event and the method name to call when the event is fired.

```
...
public:
  SqlEventHandler(SqlDataAdapter* adapter)
  {
    adapter->add_RowUpdated(
      new SqlRowUpdatedEventHandler(this,
        SqlEventHandler::OnUpdated));
  }
```

3. Implement the event-handling method. Since the `SqlRowUpdatedEventHandler` delegate defines the signature for the method, the method you define must match it perfectly. In this case, the `SqlRowUpdatedEventHandler` signature states that the method must accept two parameters: an `Object` representing the source of the event (the data adapter object) and an `SqlRowUpdatedEventArgs` object, which will be passed when the event fires and the method is called.

   Now let's look at an example method. The first thing you must do is verify that the row causing the event to be fired is a row that was added (as opposed to updated). Once you've determined that the row is a newly added row, construct a command object specifying the SQL `SELECT @@IDENTITY` command to query the data store for the new row's auto-incremented value. Calling the command object's `ExecuteScalar` method executes the command. Note that `ExecuteScalar` returns the first column of the first row of a result set that's enough for our needs, since we're only retrieving one value. With the new auto-increment value at hand, update the row's appropriate column. In the example's case, that is the `EmployeeID` column. Finally, call the `AcceptChanges` method to commit the changes to the `DataRow` object.

```
...
public:
  void OnUpdated(Object* obj, SqlRowUpdatedEventArgs* e)
  {
    if (StatementType::Insert == e->StatementType)
    {
      SqlCommand* command =
        new SqlCommand(S"SELECT @@IDENTITY",
                       e->Command->Connection);
      e->Row->Item[S"EmployeeID"] = command->ExecuteScalar());
      e->Row->AcceptChanges();
    }
  }
```

4. That's it for the event-handling side of things. Now you simply construct the new class—typically just after constructing the data adapter:

```
...
SqlConnection* conn =
```

```
    new SqlConnection(S"Server=localhost;"
                      S"Database=Northwind;"
                      S"Integrated Security=true;");
adapter = new SqlDataAdapter(S"SELECT * FROM Employees", conn);
eventHandler = new SqlEventHandler(adapter, this);
...
```

Now the `SqlEventHandler::OnUpdated` gets called for each updated or inserted row anytime you call the adapter's `Update` method to synchronize your local in-memory changes with the data store.

Filling in Missing Schema and Primary Key Information for Untyped Datasets

When using untyped datasets, certain schema information—such as primary key information is not available. The following code snippet illustrates this point. If you want to test this code, you'll need to create the second dataset (`EmployeesDataSet`) using the steps listed in the section entitled "Generating a Typed DataSet").

```
SqlConnection* conn =
  new SqlConnection(S"Server=localhost;"
                    S"Database=Northwind;"
                    S"Integrated Security=true;");
adapter* = new SqlDataAdapter(S"SELECT * FROM Employees", conn);
conn->Open();
// Untyped DataSet
DataSet* untyped = new DataSet();
adapter->Fill(untyped, S"AllEmployees");
int count1 =
  dataset->Tables->Item["AllEmployees"]->PrimaryKey->Length;
// count1 will be 0
// Typed DataSet
EmployeesDataSet* typed = new EmployeesDataSet();
adapter->Fill(typed->EmployeesDataTable);
int count2 = typed->EmployeesDataTable->PrimaryKey->Length;
// count2 will be 1
conn->Close();
```

As the comments indicate, the data table will only return the correct count of primary keys if the `DataSet` is typed. This can cause a problem if you wish to use certain ADO.NET functionality, which relies on the data

table properly representing the data store. For example, the `DataRowCollection::Find` method allows you to search through a row collection based on a primary key. However, attempting to call the `Find` method using the untyped dataset, as illustrated in the previous code snippet, results in a `System::Data::MissingPrimaryKeyException` because no primary key has been defined—at least as far as the data table object is concerned. There are two solutions to this problem.

The first solution is simply to tell the data adapter to gather the schema information when filling the dataset. This is accomplished via the data adapter's `MissingSchemaAction` property, which accepts a `MissingSchemaAction` enumeration value that tells the adapter what action to take when the `DataSet` schema doesn't match the incoming data. For example, suppose you programmatically define a `DataTable` object that doesn't match the incoming data in terms of the number of columns. Let's say you've defined a `DataTable` that has only two columns, and you want to read the `Employees` table data (which contains many more columns) into that `DataTable`. Since the schemas don't match, the data adapter has to be told how to handle that situation, and that's exactly what the `MissingSchemaAction` enumeration is typically used for. By specifying a value of `MissingSchemaAction::Add`, the adapter will add any necessary columns to complete the schema. Specifying a value of `MissingSchemaAction.Ignore` results in the data adapter ignoring any extra columns (and not downloading that data) and `MissingSchemaAction.Error` results in an exception if the schemas don't match. The default `MissingSchemaAction` property value is `Add`.

So, as you can see, the `MissingSchemaAction` property is mainly used to map dissimilar schemas to one another, which doesn't seem to be what we're after. However, there is one last `MissingSchemaAction` enumeration value—`AddWithKey`—that does exactly what we want. This value is similar to the `Add` value except that it also adds the primary key information to complete the schema. As a result, simply setting the `MissingSchemaAction` property to `AddWithKey` just before the `Fill` method call will result in the `DataTable` being properly constructed with the data source's primary key information:

```
SqlConnection* conn =
  new SqlConnection(S"Server=localhost;"
                    S"Database=Northwind;"
                    S"Integrated Security=true;");
```

```
adapter = new SqlDataAdapter(S"SELECT * FROM Employees", conn);
adapter->MissingSchemaAction = MissingSchemaAction::AddWithKey;
conn->Open();
dataset = new DataSet(); adapter->Fill(dataset, S"AllEmployees");
// Untyped DataSet
DataSet* untyped = new DataSet();
adapter->Fill(untyped, S"AllEmployees");
int count1 =
  dataset->Tables->Item["AllEmployees"]->PrimaryKey->Length;
// count1 will now be 1
```

One important thing to note is that when a data adapter whose `MissingSchemaAction` property is set to `AddWithKey` creates a `DataColumn` object for a column defined as a primary key in the data source, that `DataColumn` object is marked as read-only (`DataColumn::ReadOnly` is set to `true`). Obviously, if you do not need to alter the primary key value, this solution will work fine for you.

However, there are plenty of scenarios where you'll need to modify the local value for a primary key. For example, in the previous section you saw that the local value for an auto-increment primary key of a new row can't be realized until the row is inserted and that value is retrieved from the data store. The local value is then updated to properly reflect the data store's value. This can't be done if the `DataColumn` is set to read-only. Therefore, another mechanism must be used to indicate that a given data source column is a primary key. We can do this via the `DataTable::PrimaryKey` property.

```
SqlConnection* conn =
  new SqlConnection(S"Server=localhost;"
                    S"Database=Northwind;"
                    S"Integrated Security=true;");
adapter = new SqlDataAdapter(S"SELECT * FROM Employees", conn);
conn->Open();
dataset = new DataSet(); adapter->Fill(dataset, S"AllEmployees");
DataTableCollection* tables = dataset->Tables;
employeesTable = tables->Item[S"AllEmployees"];
DataColumn* primaryKeys[] = new DataColumn*[1];
primaryKeys[0] = employeesTable->Columns-> Item[0];
employeesTable->PrimaryKey = primaryKeys;
// Untyped DataSet
DataSet* untyped = new DataSet();
```

```
adapter->Fill(untyped, S"AllEmployees");
int count1 =
  dataset->Tables->Item["AllEmployees"]->PrimaryKey->Length;
// count1 will now be 1
```

After the `DataSet` object is filled, a `DataColumn` array is allocated, and the first `DataColumn` object from the employees `DataTable` is inserted into it, as the first column is the `EmployeeId` primary key. The `PrimaryKey` property is then set to the `DataColumn` array, thereby accomplishing what we need.

You could also set the `MissingSchemaAction` property and then, after the `DataSet` is filled, set the desired `DataColumn` object's `ReadOnly` property to false. However, it's my opinion that if you're going to manually override what the adapter does anyway, you might as well save yourself the overhead of requesting that the schema information be retrieved during the `DataSet` fill and simply override the entire process.

So, in summary, I recommend using the `MissingSchemaAction` property if your application will not need to modify the `DataColumn` object corresponding to the primary key; otherwise use the `PrimaryKey` property.

Demo—Simple Maintenance Application

Let's test what you've learned to this point with a semi-realistic MFC SDI demo application that lists all the employee records from the sample SQL Server Northwind database's Employees table. The application will allow you to create new records as well as edit and delete existing records using the classes and techniques you've learned about up to this point. While this demo contains a few more steps than I normally prefer to include in a book demo, there are a lot of little things you have to do to make a realistic ADO.NET application, and seeing how everything fits together is paramount to understanding how to use ADO.NET in a real-world application.

1. To get started, create a new MFC project called `Employee Maintenance`—where the application type is SDI and the view class is a `CListView`—and update the `Project` properties to support Managed Extensions.
2. Open the `stdafx.h` file and add the following .NET support directives to the end of the file.

```
#using <mscorlib.dll>
#using <system.dll>
```

```
#using <system.data.dll>
#using <system.xml.dll>
#using <system.windows.forms.dll>

using namespace System;
using namespace System::Data;
using namespace System::Data::SqlClient;
using namespace System::Xml;
using namespace System::Windows::Forms;
#undef MessageBox
```

3. In the view class's `PreCreateWindow` function, set the view window's style to "report" and single selection:

```
BOOL CEmployeeMaintenanceView::PreCreateWindow(CREATESTRUCT&
cs)
{
  cs.style |= LVS_REPORT;
  cs.style |= LVS_SINGLESEL;
  return CListView::PreCreateWindow(cs);
}
```

4. Add the following code to the end of the view's `OnInitialUpdate` function to initialize the list view.

```
void CEmployeeMaintenanceView::OnInitialUpdate()
{
      CListView::OnInitialUpdate();

  ...

  CListCtrl& lst = GetListCtrl();
  // All full row selection
  LONG lStyle =
  (LONG)lst.SendMessage(LVM_GETEXTENDEDLISTVIEWSTYLE);
  lStyle |= LVS_EX_FULLROWSELECT;
  lst.SendMessage(LVM_SETEXTENDEDLISTVIEWSTYLE,  0,
    (LPARAM)lStyle);
  // Add columns to listview lst.InsertColumn(0, _T("ID"));
  lst.InsertColumn(1, _T("First Name"));
  lst.InsertColumn(2, _T("Last Name"));
}
```

5. Now that the main UI is done, let's focus on the view-level ADO.NET objects and their construction. First, define the following

ADO.NET objects in the `CEmployeeMaintenanceView` class that will be used throughout the application.

```
class CEmployeeMaintenanceView : public CListView
{
...
protected:
  gcroot<DataSet*>dataset;
  gcroot<SqlDataAdapter*>adapter;
  gcroot<DataTable*>employeesTable;
  gcroot<SqlCommandBuilder*>commandBuilder;
...
```

6. Add the following code to the end of the view's `OnInitialUpdate` function to initialize the ADO.NET objects. As you can see, the first thing that is done is to make the connection to the sample SQL Server Northwind database. From there, the code constructs an `SqlDataAdapter` object (`adapter`) with an `SQL SELECT` statement that retrieves all records from the Employees table.

   Once the adapter has been constructed, an `SqlCommandBuilder` object (`eventHandler`) is instantiated, and the connection is opened. A `DataSet` object (`dataset`) is then constructed and filled with employee records via the `SqlDataAdapter::Fill` method, and the resulting `DataTable` object is named "AllEmployees." At this point, we have the data in memory, and so the connection to the data store is closed. A `DataTable` object (`employeesTable`) is then allocated and points to the "AllEmployees" `DataTable` created during the `Fill` method. (Note that declaring a `DataTable` object is done just for convenience, as the application could retrieve the table from the `DataSet` object's `DataTable` collection each time it needs access to the table.)

   The `employeesTable` primary key is defined as the table's first column via the `PrimaryKey` property. This is done so that the `DataRowCollection::Find` method can be used to locate records by their primary key. Finally, a helper function called `ReadAllEmployees` is called.

```
void CEmployeeMaintenanceView::OnInitialUpdate()
{
  CListView::OnInitialUpdate();

  ...
```

```
#pragma push_macro("new")
#undef new
try
  {
    SqlConnection* conn =
      new SqlConnection(S"Server=localhost;"
                        S"Database=Northwind;"
                        S"Integrated Security=true;");
    adapter = new SqlDataAdapter(S"SELECT * FROM Employees",
      conn);
    commandBuilder = new SqlCommandBuilder(adapter);
    conn->Open();
    dataset = new DataSet();
    adapter->Fill(dataset, S"AllEmployees");
    conn->Close(); // No longer needed
    DataTableCollection* tables = dataset->Tables;
    employeesTable = tables->Item[S"AllEmployees"];
    // Set the table's primary key column(needed for
    // DataRowCollection::Find method) Can't use
    // DataAdapter::MissingSchemaAction because that
    // would make the EmployeeID readonly and I need to
    // set that value after inserts are realized against
    // the data store.
    DataColumn* primaryKeys[] = new DataColumn*[1];
    primaryKeys[0] = employeesTable->Columns->Item[0];
    employeesTable->PrimaryKey = primaryKeys;
    employeesTable->Columns->Item[0]->ReadOnly = false;
    ReadAllEmployees();
  }
  catch(Exception* e)
  {
    MessageBox::Show(e->Message, S".NET Exception Thrown",
                     MessageBoxButtons::OK,
                     MessageBoxIcon::Error);
  }
#pragma pop_macro("new")
}
```

7. Implement the following `ReadAllEmployees` member function. After initializing the list view, this function enumerates the `employeesTable` object's row collection, retrieving each record's

EmployeeId, FirstName and LastName values and inserting them into the list view.

```
void CEmployeeMaintenanceView::ReadAllEmployees()
{
  try
  {
    CWaitCursor wc;
    CListCtrl& lst = GetListCtrl(); lst.DeleteAllItems();
    DataRowCollection* rows = employeesTable->Rows;
    DataRow* row;
    String* firstName;
    String* lastName;
    String* id;
    for (int i = 0; i < rows->Count; i++)
    {
      row = rows->Item[i];
      id = row->Item[S"EmployeeID"]->ToString();
      firstName = row->Item[S"FirstName"]->ToString();
      lastName = row->Item[S"LastName"]->ToString();
      int idx = lst.InsertItem(i, (CString)id);
      lst.SetItemText(idx, 1, (CString)firstName);
      lst.SetItemText(idx, 2, (CString)lastName);
    }
  }
  catch(Exception* e)
  {
    MessageBox::Show(e->Message, S".NET Exception Thrown",
                     MessageBoxButtons::OK,
                     MessageBoxIcon::Error);
  }
}
```

8. At this point, you should be able to build and run the application and see the employee records in the list view. Now, let's add the ability to create, edit, and delete records. Begin by adding an Employees pop-up menu as shown in Figure 6–2.

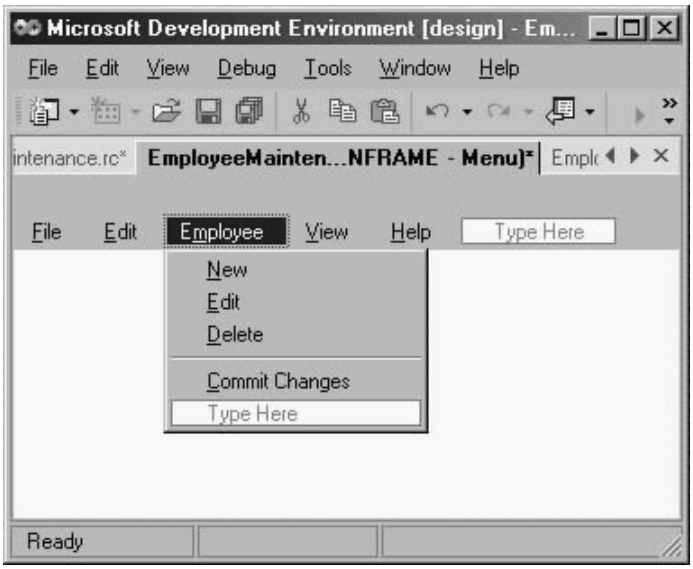

Figure 6–2 The Employee Maintenance menu for adding, editing, and deleting employee records

9. Now let's implement the ability to create new employee records. Start by adding the dialog shown in Figure 6–3.
10. Add a `CDialog`-based class called `CEmployeeDlg` for the dialog resource and include its header file in the view's implementation file (`EmployeeMaintenanceView.cpp`).

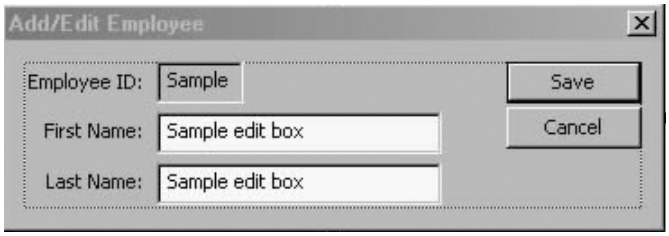

Figure 6–3 The dialog for adding and editing employee records

Table 6–1 DDX variables for the `EmployeeMaintenance` demo

Control	Variable Type	Variable Name
Employee ID	`int`	`m_iEmployeeId`
First Name	`Cstring`	`m_strFirstName`
Last Name	`Cstring`	`m_strLastName`

11. Returning to the resource editor, add the DDX value variables for the `CEmployeeDlg` as shown in Table 6–1.
12. Each employee record is uniquely identified by its `EmployeeId` column. However, as this column is defined as auto-increment (`IDENTITY`), we won't know its value for a newly created record until the data source has been updated, the database has generated the value, and that value is retrieved using a technique such as that described in the section entitled "Disconnected Data and Auto-Increment Primary Keys." However, we need a way to uniquely identify newly created records before the data store is updated. Therefore, we need to associate any newly created records with an application-generated key that will not conflict with any possible real values. One way to accomplish this is to create an application variable that starts out as a negative value and decrements with each new record. This will work because database auto-increment values start at 0 and increment. Therefore, define the following static variable (`newRowId`) to the top of the EmployeeMaintenanceView.cpp file.

```
// Used as a temporary id for new records not yet added to the
database
static newRowId = -1;
```

13. The next thing we'll want to do is to track which records in the list view are new records. We'll do this so that when the data store is updated, it's easy to locate the affected records and update their `EmployeeId` value with the data store generated value. We'll use a simple MFC map collection to do this. Define the following `CMapPtrToPtr` member in the view class.

```
class CEmployeeMaintenanceView : public CListView
{
...
```

```
protected:
  CMapPtrToPtr mapIdToLVIndex;
```

14. Now that everything's in place, add the following `#define` directive and event handler for the `New` option of the Employee menu to the `CEmployeeMaintenanceView` class. The function first displays the `CEmployeeDlg`. If the user clicks the Save (IDOK) button, the function constructs a new `DataRow` object and initializes it by setting the `EmployeeId` to the value of the `newRodId` and the `First Name` and `LastName` values to those of the dialog's member variables (representing the data entered by the user).

The data is then added to the list view. You'll notice that I opted to set the list view's `employeeID` column to `NEW_RECORD ("NEW")` instead of the `newRowId` value. This was simply a choice of aesthetics. However, since I need to keep track of the row's `newRowId` value, I stuff that value into the item's item data for later retrieval. The map is then updated where the `newRowId` maps to the index of the newly added item in the list view. Finally, the `newRowId` value is decremented for the next new record.

```
#define NEW_RECORD "NEW"

...

void CEmployeeMaintenanceView::OnEmployeeNew()
{
  try
  {
    CListCtrl& lst = GetListCtrl();
    CEmployeeDlg dlg; if (IDOK == dlg.DoModal())
    {
      CWaitCursor wc;
      DataRow* newRow = employeesTable->NewRow();
      newRow->Item[S"EmployeeID"] = __box(newRowId);
      newRow->Item[S"FirstName"] =
        (String*)dlg.m_strFirstName;
      newRow->Item[S"LastName"] = (String*)dlg.m_strLastName;
      employeesTable->Rows->Add(newRow);
      int idx = lst.InsertItem(lst.GetItemCount(),
        NEW_RECORD);
      lst.SetItemData(idx, (int)newRowId);
      lst.SetItemText(idx, 1, dlg.m_strFirstName);
      lst.SetItemText(idx, 2, dlg.m_strLastName);
```

```
        mapIdToLVIndex.SetAt((void*)newRowId, (void*)idx);
        // when finished, decrement static new row counter
        newRowId--;
      }
    }
    catch(Exception* e)
    {
      MessageBox::Show(e->Message, S".NET Exception Thrown",
                       MessageBoxButtons::OK,
                       MessageBoxIcon::Error);
    }
}
```

15. We can add records to the list view, but that's all. Therefore, let's implement the Commit Changes menu item so that we can see the data store get updated. As you can see, there's really nothing much to do here besides call the data adapter's `Update` method—specifying which table to update—and handle any potential exceptions.

```
void CEmployeeMaintenanceView::OnEmployeeCommitChanges()
{
  try
  {
    CWaitCursor wc;
    adapter->Update(employeesTable);
    MessageBox::Show(S"Changed committed", S"Information",
                     MessageBoxButtons::OK,
                     MessageBoxIcon::Information);
  }
  catch(Exception* e)
  {
    MessageBox::Show(e->Message, S".NET Exception Thrown",
                     MessageBoxButtons::OK,
                     MessageBoxIcon::Error);
  }
}
```

16. At this point, building and running the code results in new records being added to the data store. However, we still have one hurdle to clear. We need to update the `DataRow` (and list view) with the data store generated `EmployeeId`. If we skip this, attempts to edit or delete the row after updating the data store will result in an exception, as our application-supplied temporary value is invalid. To start

with, forward-declare the following class before the `CEmployeeMaintenanceView` class. We have to do this because—as you'll soon see—there's a circular reference between the `SqlEventHandler` class and the `CEmployeeMaintenanceView` class.

```
__gc class SqlEventHandler;
class CEmployeeMaintenanceView : public CListView
{
...
```

17. Now, implement the `SqlEventHandler` class. This class will be used to handle the data adapter's `RowUpdated` event so that we can determine the new `EmployeeId` value for newly inserted records. The class's constructor takes both a pointer to an adapter object as well as a pointer to the view object. The latter is needed because the object needs to tell the view when it has handled an event. The `OnUpdated` method—what actually gets called when the data adapter's `RowUpdated` event fires—saves the previous `EmployeeId` value and then uses the `SELECT @@IDENTITY` command to retrieve the new `EmployeeId` value. Both values are then passed to the view's `OnRowInserted` function.

```
__gc class SqlEventHandler
{
public:
  SqlEventHandler(SqlDataAdapter* adapter,
                  CEmployeeMaintenanceView* parentView)
  {
    adapter->add_RowUpdated(new
      SqlRowUpdatedEventHandler(this,
      SqlEventHandler::OnUpdated));
    this->parentView = parentView;
  }
protected:
  CEmployeeMaintenanceView* parentView;
public:
  void OnUpdated(Object* obj, SqlRowUpdatedEventArgs* e)
  {
    if (StatementType::Insert == e->StatementType)
    {
      SqlCommand* command =
        new SqlCommand(S"SELECT @@IDENTITY",
                       e->Command->Connection);
```

```
      int previousId =
        *dynamic_cast<__box int*>(e->Row->
          Item[S"EmployeeId"]);
      e->Row->Item[S"EmployeeID"] = command->ExecuteScalar();
      int newId =
        *dynamic_cast<__box int*>(e->Row->
          Item[S"EmployeeId"]);
      e->Row->AcceptChanges();
      parentView->OnRowInserted(previousId, newId);
    }
  }
};
```

18. Now, implement the view's (public) `OnRowInserted` function as follows. Here the function is simply using the previous `EmployeeId` value to search the `mapIdToLVIndex` collection for the list view index of the just-inserted row. The list view item is then updated with the new `EmployeeId` value, and the entry is removed from the `mapIdToLVIndex` collection, as this is the only function that uses it, and it's no longer needed once the `EmployeeId` is updated.

```
void CEmployeeMaintenanceView::OnRowInserted(int iPrevId, int
iNewId)
{
  void* iListViewIdx;
  mapIdToLVIndex.Lookup((void*)iPrevId, (void*&)iListViewIdx);
  ASSERT(-1 < (int)iListViewIdx); if (-1 < (int)iListViewIdx)
  {
    CListCtrl& lst = GetListCtrl();
    CString strNewId;
    strNewId.Format(_T("%ld"), iNewId);
    lst.SetItemText((int)iListViewIdx, 0, strNewId);
    mapIdToLVIndex.RemoveKey((void*)iPrevId);
  }
}
```

19. Define a `CEmployeeMaintenanceView` member variable called `eventHandler` that is of type `SqlEventHandler` This is why we had to forward-declare the `SqlEventHandler` class before the view class's definition.

```
class CEmployeeMaintenanceView : public CListView
{
...
```

```
public:
  gcroot<SqlEventHandler*>eventHandler;
```

20. Now that the code is in place to handle the data adapter's `RowUpdated` event, we need only subscribe to the event. Since the subscription to the event takes place in the `SqlEventHandler` object's construct, we just need to construct the `SqlEventHandler` at the appropriate time. The perfect place to do this is in the view's `OnInitialUpdate` function immediately after the data adapter object has been instantiated:

```
void CEmployeeMaintenanceView::OnInitialUpdate()
{
...
adapter = new SqlDataAdapter(S"SELECT * FROM Employees",
  conn);
eventHandler = new SqlEventHandler(adapter, this);
```

21. At this point, you can now add records to the data store! Let's finish up this demo by implementing the edit and delete functions. Start by adding the following helper function to the `CEmployeeMaintenanceView` class, which will return the currently selected item index of the list view.

```
int CEmployeeMaintenanceView::GetSelectedItem()
{
  int iCurrSel = -1;
  CListCtrl& lst = GetListCtrl();
POSITION pos = lst.GetFirstSelectedItemPosition();
  if (pos)
    iCurrSel = lst.GetNextSelectedItem(pos);
  return iCurrSel;
}
```

22. Implement the following event handler for the Edit option of the Employee menu. As you can see, the function attempts to determine the `EmployeeId` for the row by first looking at the first column of the list view. However, if that value is equal to `"NEW"`, then the function retrieves the `EmployeeId` value from the list view item's item data. From there, the `CEmployeeDlg` object's member variables are initialized, and the dialog is displayed.

If the user enters data and clicks the Save button (IDOK), the `DataRow` for the edited row is located by specifying the `EmployeeId` to the `DataRowCollection::Find` method. Being

able to search the `DataRowCollection` is another reason why the application needed to maintain a temporary `EmployeeId` for new rows. Once the row is retrieved, it's `FirstName` and `LastName` columns are updated and so is the list view.

```
void CEmployeeMaintenanceView::OnEmployeeEdit()
{
  try
  {
    CListCtrl& lst = GetListCtrl();

    int currSel = GetSelectedItem();
    if (-1 < currSel)
  {
    CEmployeeDlg dlg;

    CString strId = lst.GetItemText(currSel, 0);
    Int32 id;
    if (0 == strId.Compare(NEW_RECORD))
      id = (int)lst.GetItemData(currSel);
    else
      id = atoi(strId);

     dlg.m_iEmployeeId = id;
     dlg.m_strFirstName = lst.GetItemText(currSel, 1);
     dlg.m_strLastName = lst.GetItemText(currSel, 2);

    if (IDOK == dlg.DoModal())
    {
      CWaitCursor wc;
      DataRow* row = employeesTable->Rows->Find(__box(id));
      if (row)
      {
        row->Item[S"FirstName"] = (String*)dlg.m_strFirstName;
        row->Item[S"LastName"] = (String*)dlg.m_strLastName;

        lst.SetItemText(currSel, 1, dlg.m_strFirstName);
        lst.SetItemText(currSel, 2, dlg.m_strLastName);
      }
    }
  }
  else
  {
    MessageBox::Show(S"You must first select an employee "
```

```
            S"to perform this operation.",
            S"Alert",
            MessageBoxButtons::OK,
            MessageBoxIcon::Error);
  }
}
catch(Exception* e)
{
  MessageBox::Show(e->Message, S".NET Exception Thrown",
           MessageBoxButtons::OK,
           MessageBoxIcon::Error);
  }
}
```

23. Finally, implement the following event handler for the Delete option of the Employee menu. The function starts out much like the Edit menu event handler by first retrieving the `DataRow` for the currently selected employee. Once that is done, the `DataRow::Delete` method is called, and the row is removed from the list view.

```
void CEmployeeMaintenanceView::OnEmployeeDelete()
{
  try
  {
    CListCtrl& lst = GetListCtrl();
    int currSel = GetSelectedItem(); if (-1 < currSel)
    {
      CWaitCursor wc;
      CString strId = lst.GetItemText(currSel, 0);
      Int32 id;
      if (0 == strId.Compare(NEW_RECORD))
        id = (int)lst.GetItemData(currSel);
      else
        id = atoi(strId);
      DataRow* row = employeesTable->Rows->Find(__box(id));
      if (row)
      {
        row->Delete();
        lst.DeleteItem(currSel);
      }
    }
    else
    {
```

```
        MessageBox::Show(S"You must first select an employee "
                         S"to perform this operation.",
                         S"Alert",
                         MessageBoxButtons::OK,
                         MessageBoxIcon::Error);
      }
    }
    catch(Exception* e)
    {
      MessageBox::Show(e->Message, S".NET Exception Thrown",
                       MessageBoxButtons::OK,
                       MessageBoxIcon::Error);
    }
  }
```

Finally we're done! As I mentioned in the beginning of this chapter, while disconnected data is a wonderful thing for certain scenarios, it also increases the complexity of even something as simple as a maintenance application. However, just saying (or writing) that doesn't have the same impact as actually walking through a step-by-step demo such as this `EmployeeMaintenance` application and seeing for yourself the various issues that arise in a disconnected setting and how to deal with them one by one. Figure 6–4 shows an example of the application being run.

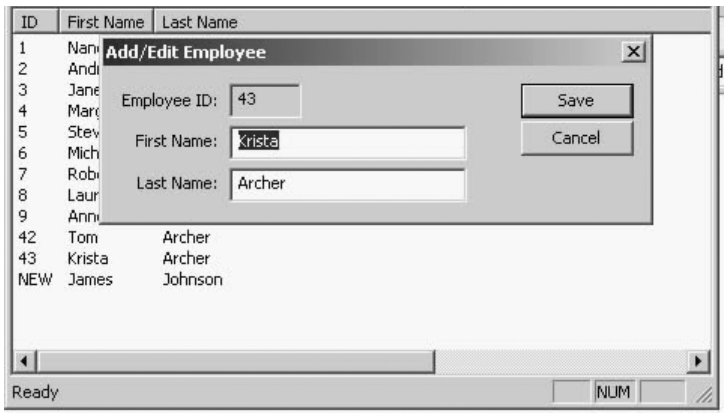

Figure 6–4 The `EmployeeMaintenance` demo at work

Working in Batch

There are times when it's advantageous to perform certain operations in batch. For example, multiple read operations can be performed with a single `Fill` method, thereby causing the generation of more than one local `DataTable` while incurring the cost of only one round trip between the client and the server. In addition to that, this section will also illustrate how to turn off `DataTable` constraints so that multiple changes can be made to a `DataRow` in batch and then either all accepted or canceled with a single method call. Finally, we'll also look at a way to turn index maintenance off to improve the performance associated with loading large amounts of data into a `DataTable`.

Creating Multiple `DataTable`s in a `DataSet`

The principal benefits of using disconnected data are to prevent sustained client connections to the data source and to minimize round trips between client application(s) and the data server. To that end, there are times when it's advisable to batch your reads or queries against the data source so that you can fill more than one `DataTable` with a single round trip. Take the following example, where I'm creating a `DataSet` object with two `DataTable` objects—one filled with records from the Employees table and one filled with records from the Products table.

```
void MultipleRoundTrips()
{
#pragma push_macro("new")
#undef new
  try
  {
    SqlConnection* conn =
      new SqlConnection(S"Server=localhost;"
                        S"Database=Northwind;"
                        S"Integrated Security=true;");
    // Construct the employees data adapter
    SqlDataAdapter* employeesAdapter =
      new SqlDataAdapter(S"SELECT * FROM Employees", conn);
    // Construct the products data adapter
    SqlDataAdapter* productsAdapter =
```

```
      new SqlDataAdapter(S"SELECT * FROM Products", conn);
    conn->Open();
    DataSet* dataset = new DataSet();
    // Get all the employees - FIRST TRIP TO SERVER
    employeesAdapter->Fill(dataset, S"AllEmployees");
    // Get all the products - SECOND TRIP TO SERVER
    productsAdapter->Fill(dataset, S"AllProducts");
    conn->Close(); // No longer needed
    DataTableCollection* tables = dataset->Tables;
    DataTable* table;
    for (int i = 0; i < tables->Count; i++)
    {
      table = tables->Item[i];
      Console::WriteLine(String::Format(S"{0} table has {1} rows",
                         table->TableName,
                         __box(table->Rows->Count)));
    }
  }
  catch(Exception* e)
  {
    Console::WriteLine(e->Message);
  }
#pragma pop_macro("new")
}
```

As the comments indicate, two round trips are being made here—one for each `DataTable`. In situations like this—where you have all the information you need to specify your `SELECT` statement—it's more efficient to specify a `SELECT` statement in the data adapter's constructor that will cause each table to be retrieved in single round trip. Simply delimiting each `SELECT` statement with a semicolon accomplishes this:

```
String* allEmployees = S"SELECT * FROM Employees";
String* allProducts = S"SELECT * FROM Products";
String* select = String::Format(S"{0};{1}", allEmployees,
                                allProducts);
SqlDataAdapter* adapter = new SqlDataAdapter(select, conn);
```

At this point, you might be wondering how to name the `DataTable` when using this technique. After all, the `Fill` method takes the name of a

DataTable, but now we're reading two tables. The following function answers that question:

```
void BatchRead()
{
#pragma push_macro("new")
#undef new
  try
  {
    SqlConnection* conn =
      new SqlConnection(S"Server=localhost;"
                        S"Database=Northwind;"
                        S"Integrated Security=true;");
    String* allEmployees = S"SELECT * FROM Employees";
    String* allProducts = S"SELECT * FROM Products";
    String* select = String::Format(S"{0};{1}",
                                    allEmployees,
                                    allProducts);
    // Read two tables at once to reduce round trips
    SqlDataAdapter* adapter = new SqlDataAdapter(select, conn);
    conn->Open();
    DataSet* dataset = new DataSet();
    // Fill dataset. Don't bother naming the table because
    // the second table has to be renamed anyway.
    adapter->Fill(dataset);
    conn->Close(); // No longer needed
    DataTableCollection* tables = dataset->Tables;
    DataTable* table;
    for (int i = 0; i < tables->Count; i++)
    {
      table = tables->Item[i];
      Console::WriteLine(table->TableName);
    }

    tables->Item[0]->TableName = S"AllEmployees";
    tables->Item[1]->TableName = S"AllProducts";
    for (int i = 0; i < tables->Count; i++)
    {
      table = tables->Item[i];
      Console::WriteLine(String::Format(S"{0} table has {1} rows",
                        table->TableName,
                        __box(table->Rows->Count)));
    }
```

```
  }
  catch(Exception* e)
  {
    Console::WriteLine(e->Message);
  }
#pragma pop_macro("new")
}
```

This function produces the following output:

```
Table
Table1
AllEmployees table has 11 rows
AllProducts table has 77 rows
```

As mentioned earlier in the chapter, if you call the `Fill` method and do not specify a name for the `DataTable` that will be generated, it defaults to Table. If more than one `DataTable` is created, they are named Table1, Table2, and so on, as you can see in the first two lines of the output. However, what happens if you do specify a name? In this case, the first table gets the specified name and the remaining tables are simply named the same thing with a sequential numeric value (starting with 1) affixed to the end. Therefore, in order to have symbolic `DataTable` names, you'll need to set the `DataTable::TableName` property manually after a read that returns more than one table. As the last two lines of the output indicate, the two `DataTable` objects were renamed, and their row counts reflect the fact that we did indeed perform two separate queries with one round trip.

Batch Row Changes

The term "batch row changes" or "batch row updates" can be a bit misleading; it simply refers to the ability to change multiple properties (including column values) of a `DataRow` object before committing those changes. This can be extremely useful in several scenarios. For example, because of certain constraints on a given table, you might need to make two or more changes to a row before the row is valid, yet the first change results in an exception because you are in violation of the constraint. Let's say you're working with the Employees table and you want to store either a location to the employee's picture (such as a URL) or the actual binary data that comprises the picture. Since these are two drastically different data types (string and binary, respectively) you could create a column for each possibility and

set up a constraint that states that one of the columns must have a value and the other must be null—your standard XOR situation. Now let's say that you add a record with the picture data column set and the picture location column null, but later want to reverse that. Because of the aforementioned constraint, this would be extremely difficult.

As another practical example of needing to process multiple row changes in parallel, or batch, suppose you had a wizard-like interface that allowed the user to modify different columns of a row through a series of dialogs. As the user might cancel the operation at any point, you would need the ability to reverse the changes the user did make to the `DataRow`.

Both of these scenarios can be handled very simply by using the `DataRow` class's `BeginEdit`, `EndEdit`, and `CancelEdit` methods.

When the `BeginEdit` method is called, all events are temporarily suspended for the row so that any changes made do not trigger validation rules (constraints). Calling `EndEdit` then commits these changes and brings back into play any events and constraints defined for the table/row. If you decide that you do not want to commit the changes, you can simply call the `CancelEdit` method and any changes made to the row since the `BeginEdit` method was called will be lost. Using the picture example from above, the code to change the values could now be written as follows:

```
// row is a DataRow having a column for picture data,
// which is currently set and a column for picture location,
// which needs to be set - all without breaking the constraint
// that one of these columns needs to be null at all times
// Start the edit process - turning off constraints
row->BeginEdit();
// Make changes row->Item[S"PictureData"] = DBNull::Value;
row->Item[S"PictureLocation"] = S"http://somelocation.com";
// Commit the changes
row->EndEdit();
```

Batch Table Updates

Much as the `DataRow` class allows you to suspend validation rules in order to perform certain operations in batch, the `DataTable` also provides the same capability at the table level, using the `BeginLoadData` and `EndLoadData` method pair. In fact, because we are talking about the table level, index maintenance is also suspended, providing a much faster means of loading already validated data into a `DataTable`. In fact, my own benchmarking—using some of the performance and benchmarking techniques

discussed in Chapter 10—indicate up to a 30% increase in speed simply by calling `BeginLoadData`, loading the rows via `LoadDataRow`, and then calling `EndLoadData` when finished!

As an example of a realistic situation where you'd want to use these methods, let's say you were loading a large amount of data into a `DataTable` from another data source—such as a text file. Listing 6–1 shows the partial listing of the `rockets.txt` file included on the book's CD for the `BatchTableLoad` demo application.

Listing 6–1 Partial listing of sample comma-delimited text file (`rockets.txt`)

```
"Yao","Ming",13.5,"Dynasty"
"Steve","Francis",21.0,"Franchise"
"Cuttino","Mobley",17.5,"Cat"
...
```

Now let's see how we could read this data and batch load it into a `DataTable`.

```
void BatchLoad()
{
#pragma push_macro("new")
#undef new
  try
  {
    // Define a DataTable
    DataSet* dataset = new DataSet();
    DataTable* table = new DataTable(S"Players");
    dataset->Tables->Add(table);
    // Define the table's columns
    table->Columns->Add(S"FirstName", __typeof(String));
    table->Columns->Add(S"LastName", __typeof(String));
    table->Columns->Add(S"PointsPerGame", __typeof(Double));
    table->Columns->Add(S"Nickname", __typeof(String));
    // Open the csv file and read entire file into a String object
    StreamReader* reader = new StreamReader(S"rockets.txt");
    String* rocketStats = reader->ReadToEnd();
    reader->Close();
    String* pattern = S"\"(?<fname>[^\"]+)\","
                      S"\"(?<lname>[^\"]+)\","
                      S"(?<ppg>[^,]+),"
```

```
                    S"\"(?<nickname>[^\"]+)\"";
    Regex* rx = new Regex(pattern);
    // Disable notifications, constraints and index maintenance
    table->BeginLoadData();
    // for all the matches
    for (Match* match = rx->Match(rocketStats);
        match->Success;
        match = match->NextMatch())
    {
      Object* columns[] = new Object*[table->Columns->Count];
       GroupCollection* groups = match->Groups;
      columns[0] = groups->Item[S"fname"]->Value;
      columns[1] = groups->Item[S"lname"]->Value;
      columns[2] = groups->Item[S"ppg"]->Value;
      columns[3] = groups->Item[S"nickname"]->Value;
      // Load all fields with one call
      table->LoadDataRow(columns, true);
    }

    // Re-enable notifications, constraints and index maintenance
    table->EndLoadData();
  }
  catch(Exception* e)
  {
    Console::WriteLine(e->Message);
  }
#pragma pop_macro("new")
}
```

Once the `DataTable` is constructed and its schema defined, the text file is read using the `StreamReader` class covered in Chapter 3. Specifically, the `ReadToEnd` method is called so that the entire file is read into a `String` object. At that point, a regular expression is used, in which each `Match` object represents a row in the text file and each named `Group` object within the `Match` object represents a column of that row. (Regular expressions, matches and groups are covered in Chapter 2.)

Now that we have the data in a format that can be easily enumerated, the function calls `BeginLoadData` to disable notifications, constraints, and index maintenance while the rows are loaded into the `DataTable`. This loading is done via a call to the `LoadDataRow` method. As you can see, the first parameter to this method is an array of objects, each of which represents a column.

The second parameter is a Boolean value that indicates if the `DataTable::AcceptChanges` should be called after the table is loaded.

There are also a couple of other things you should note here. First, if a column in the `DataRow` is auto-generated, or if you want the default value used, you should simply set the object corresponding to that column to `System::Object::Empty`. However, keep in mind that the new row will only be appended to the `DataTable` if the key field does not already exist. If you're attempting to add a row with a duplicate primary key, the first row will be overwritten with the second. Therefore, while I didn't use a primary key in this example in order to keep things simple (and since I've already shown how to use primary keys in previous examples), you should take care to avoid duplicate keys. Finally, when the data has been loaded, a call to `EndLoadData` re-enables the `DataTable` object's notifications, constraints, and index maintenance.

While the batch loading of records is the most common use for the `BeginLoadData` and `EndLoadData` methods, they are also used to handle another common problem—that of swapping the primary keys of two rows. In fact, I've seen numerous times on various newsgroups that someone has attempted to use the `DataRow` class's `BeginEdit` and `EndEdit` methods to accomplish this to no avail.

```
// Disable constraint checking for both rows
row1->BeginEdit();
row2->BeginEdit();

Int32 temp = *dynamic_cast<__box int*>(row1->Item[S"ID"]);
row1->Item[S"ID"] = row2->Item[S"ID"];
row2->Item[S"ID"] = __box(temp);
row1->EndEdit();  // ERROR! - WILL FAIL HERE!
row2->EndEdit();
```

As the comment indicates, this technique will fail. The calls to the `BeginEdit` method for the two `DataRow` objects result in the disabling of constraints for the respective rows, as we want. However, once `EndEdit` is called for the first `DataRow`, the primary key value for that first row conflicts with the primary key value for the not-yet-committed second `DataRow` object. Once again, the `BeginLoadData` and `EndLoadData` methods save the day, as illustrated in the following example:

```
void SwapPrimaryKeyValues()
{
```

```
#pragma push_macro("new")
#undef new
  try
  {
    DataSet* dataset = new DataSet();
    DataTable* table = new DataTable(S"Players");
    dataset->Tables->Add(table);
    table->Columns->Add(S"ID", __typeof(Int32));
    table->Columns->Add(S"Name", __typeof(String));
    DataColumn* primaryKeys[] = new DataColumn*[1];
    primaryKeys[0] = table->Columns->Item[0];
    table->PrimaryKey = primaryKeys;
    // Create row #1
    DataRow* row1 = table->NewRow();
    row1->Item[S"ID"] = __box(1);
    row1->Item[S"Name"] = S"Foo";
    table->Rows->Add(row1);
    // Create row #2
    DataRow* row2 = table->NewRow();
    row2->Item[S"ID"] = __box(2);
    row2->Item[S"Name"] = S"Bar";
    table->Rows->Add(row2);
    // Turn off notifications, constraints and
    // index maintenance.
    table->BeginLoadData();
    // Pull the primary key switch-a-roo while nobody's
    // watching.
    Int32 temp = *dynamic_cast<__box int*>(row1->Item[S"ID"]);
    row1->Item[S"ID"] = row2->Item[S"ID"];
    row2->Item[S"ID"] = __box(temp);
    // Turn back on notifications, constraints and
    // index maintenance.
    table->EndLoadData();
  }
  catch(Exception* e)
  {
    Console::WriteLine(e->Message);
  }
#pragma pop_macro("new")
}
```

Working with Row State Information

As mentioned earlier in the chapter and illustrated in the section entitled "Deleting and Removing Rows," the `DataRow` class defines a property called `RowState` that is a `DataRowState` enumeration with the values shown in Table 6–2.

Table 6–2 `DataRowState` Enumeration Values

Member Name	Description
`Added`	The `DataRow` object has been added to the `DataRowCollection` object, but `DataRow::AcceptChanges` has not been called.
`Deleted`	The `DataRow` object—belonging to a `DataRowCollection`—has been deleted using the `DataRow::Delete` method.
`Detached`	Either the `DataRow` object has not been added to the collection or it has been removed via either the `DataRowCollection::Remove` or `DataRowCollection::RemoveAt` method.
`Modified`	The `DataRow` object—belonging to a `DataRowCollection`—has been edited, but `DataRow::AcceptChanges` has not been called.
`Unchanged`	The `DataRow` object—belonging to a `DataRowCollection`—has not changed since the last time `DataRow::AcceptChanges` was called.

As you can you can see, these enumeration values cover the various states of a `DataRow`. In fact, you can use the Visual Studio .NET debugger with the `EmployeeMaintenance` demo application and easily see the various states of the respective `DataRow` objects as you add, edit, and delete employee records. The following example illustrates the manipulation of a `DataRow` object and the impact of those changes on the object's `RowState` property. (As you've already seen examples of connecting to and reading existing data into `DataSet` objects, this code also illustrates how to manually define your own `DataSet` and `DataTable` objects from application data. It's not something you'll use every day, but it's definitely useful at times.)

```
void PrintRowState(String* currentAction, DataRow* row)
{
```

```
  Console::WriteLine(String::Format(S"{0,-20}\t\tDataRowState::{1}",
                     currentAction,
                     __box(row->RowState)));
}

void CreateAndQueryRowStates()
{
#pragma push_macro("new")
#undef new
  try
  {
    // Define a DataSet
    DataSet* dataset = new DataSet();
    DataTable* table = new DataTable(S"Players");
    dataset->Tables->Add(table);

    table->Columns->Add(S"ID", __typeof(Guid));
    table->Columns->Add(S"FirstName", __typeof(String));
    table->Columns->Add(S"LastName", __typeof(String));
    table->Columns->Add(S"Nickname", __typeof(String));
    // Create new row
    DataRow* row = table->NewRow();
    row->Item[S"ID"] = Guid::NewGuid().ToString();
    row->Item[S"FirstName"] = S"Yao";
    row->Item[S"LastName"] = S"Ming";
    row->Item[S"Nickname"] = S"";
    PrintRowState(S"Created new row", row);
    // Add row table->Rows->Add(row);
    PrintRowState(S"Added new row", row);
    // Accept changes         row->AcceptChanges();
    PrintRowState(S"Accepted changes", row);
    // Edit row
    row->Item[S"Nickname"] = S"Dynasty";
    PrintRowState(S"Edited row", row);

    // Delete row
    row->Delete();
    PrintRowState(S"Deleted row", row);
  }
  catch(Exception* e)
  {
    Console::WriteLine(e->Message);
  }
```

```
#pragma pop_macro("new")
}
```

Plugging these two functions into a Win32 console application with Managed Extensions support and the appropriate using directives would result in the following output:

```
Created new row          DataRowState::Detached
Added new row            DataRowState::Added
Accepted changes         DataRowState::Unchanged
Edited row               DataRowState::Modified
Deleted row              DataRowState::Deleted
```

While most of this is what you'd expect after seeing the `DataRowState` enumeration descriptions in Table 6–2, there are a couple of things of note. First, a newly created `DataRow` has its `RowState` set to `Detached` until it is added to the `DataRowCollection` via the `DataRowcollection::Add` method. At that point, the `RowState` is initialized to `Added` until the `DataRow::AcceptChanges` method is called, at which point the `RowState` becomes `Unchanged`. Finally, note that if you call `DataRow::Delete`, the row's `RowState` becomes `Deleted`, and subsequently calling `DataRow Collection::Remove` (or `DataRowCollection::RemoveAt`) will not affect its `RowState`. The opposite is also true: Removing a `DataRow` from a `Data RowCollection` sets the row's `RowState` to `Detached`, but subsequently deleting it will not alter its `RowState` value.

Searching, Sorting, and Filtering Data

So far, you've learned how to connect to a data store, retrieve the desired data, close the connection, modify the data locally (including adding, editing, and deleting records), and then update the data store at will—once again only holding the connection long enough to synchronize your in-memory representation of the data with the data store. As you'll discover in this section, the ADO.NET classes also support the ability to search, sort, and filter data once it's been retrieved. In fact, being able to perform these operations against your disconnected data without making continual round trips to the server is one of the strongest arguments for using disconnected data to begin with.

We'll begin by looking at how to search the `DataTable` and `DataRowCollection` objects using both primary keys and column data. After that, you'll see how to create `DataView` objects to represent different views on the same `DataTable` object where you can filter on both column values and row state information, which we covered in the previous section. Finally, you'll discover how to sort and search through these filtered `DataView` objects.

Searching `DataTable` and `DataRowCollection` Objects

There are two basic ways to search a `DataTable` and its internal `DataRowCollection`—by primary key and by column value. We'll start with the primary key search, as that technique was used in the `EmployeeMaintenance` demo application.

Searching on Primary Key

The `DataRowCollection::Find` method is used to retrieve a `DataRow` from a `DataRowCollection` by its primary key value. This method has overloaded versions that return a `DataRow` instance. A value of `null` is returned if a row containing the specified primary key value(s) cannot be located in the collection.

```
public: DataRow* Find(Object*);
public: DataRow* Find(Object*[]);
```

The first `Find` method enables you to specify a single primary key value, while the second enables you to specify an array of values if the primary key has more than one column. Note that passing an array whose length is greater than the number of defined primary keys columns results in an `ArgumentException`, where exception object's `Message` property will indicate both how many values were expected and how many were specified in the `Find` method.

Also keep in mind that in order to use the `Find` method with an untyped dataset, you'll need to either set the data adapter object's `MissingSchemaAction` property to `MissingSchemaAction::AddWithKey` (before the call to the adapter's `Fill` method) or manually set the `DataTable` object's `Primary` key property by passing it an array of `DataColumn` objects that correspond to the table's primary key columns. Both of these techniques are described in more detail in the section entitled "Filling in Missing Schema and Primary Key Information for Untyped Datasets."

In the following example, I'm using the `Find` method to locate an employee record with a primary key value of 1. As the Employees table's `EmployeeID` column is defined as its only primary key column, then this search is basically the equivalent of an SQL statement like `SELECT * where EmployeeID = 1`.

```
// Search for a primary key made up of a single column
// Assumes a connection to the Employees table
...

DataRow* row = employeesTable->Rows->Find(__box(1));
if (row)
{
  // Have DataRow to use
  Console::WriteLine(String::Format(S"Found record --> {0} {1}",
                                    row->Item[S"FirstName"],
                                    row->Item[S"LastName"]));
}
else
  Console::WriteLine(S"Could not locate record");
```

The sample Employee table contains a record for a "Nancy Davolio" that has an `EmployeeID` value of 1, so her name should print out. Now let's see how to perform a `Find` operation with more than one primary key column. As the Employees table has only a single column defined as its primary key column, the following sample will also include code to override that fact and tell the `DataTable` that the `FirstName` and `LastName` columns are now the primary key columns.

```
// Search for a primary key made up of multiple columns
// Assumes a connection to the Employees table
...

// Tell the DataTable that the FirstName and LastName
// columns are now the primary key columns
employeesTable->PrimaryKey = NULL;
DataColumn* keys[] = new DataColumn*[2];
keys[0] = employeesTable->Columns->Item[S"FirstName"];
keys[1] = employeesTable->Columns->Item[S"LastName"];
employeesTable->PrimaryKey = keys;
// Build the primary key search value array
Object* search[] = new Object*[2];
```

```
search[0] = S"Nancy";
search[1] = S"Davolio";

DataRow* row = employeesTable->Rows->Find(search);
if (row)
{
  // Have DataRow to use
  Console::WriteLine(String::Format(S"Found record --> {0}",
                                    row->Item[S"EmployeeId"]));
}
else
  Console::WriteLine(S"Could not locate record");
```

In addition to the `Find` method, the `DataRowCollection` also includes a method called `Contains`, which allows you to perform the same exact search as the `Find` method, with the difference being that instead of a `DataRow` being returned, a Boolean value is returned indicating the success or failure of the search.

Searching (and Sorting) on Column Values

In order to search a `DataTable` for a given column value, you can use the `DataTable::Select` method and specify search criteria much like those of the `SQL SELECT` statement. The `Select` method has the following overloads.

```
DataRow* Select() [];
DataRow* Select(String* criteria) [];
DataRow* Select(String* criteria, String* sort) [];
DataRow* Select(String* criteria, String* sort, DataViewRowState) [];
```

The first (parameter-less) version of this method simply returns all rows in the `DataTable` object's `DataRowCollection` in a `DataRow` array sorted by primary key. If the table does not contain a primary key, then the rows are sorted in arrival sequence (the order in which they were added to the collection).

The second—and most used—version allows you to specify the search criteria. As mentioned, this criterion is much like specifying the value of an SQL `SELECT` statement without the SQL verbs. As with the parameter-less `Select` method, the resulting `DataRow` array is either sorted by primary key or, in the absence of a defined primary key column, arrival sequence.

The third version of the `Select` method enables you to specify search criteria as well as the columns to use in sorting the returned `DataRow` array. You can also append the standard SQL ASC and DESC to the column to indicate an ascending or descending sort, respectively.

Finally, the last `Select` method overload enables you to specify search criteria using the sort columns and a `DataViewRowState` enumeration value. In the section entitled "Working with Row State Information," you learned that the `DataRow` object maintains a state property indicating such things as whether or not the row has been added or changed (but not committed), deleted, or unchanged. The valid `DataViewRowState` enumeration values and their descriptions are listed and explained in the section entitled "Filtering on Row State Information."

Here are several examples of how to use the `Select` method that illustrate how robust this feature is. For example, aside from the standard comparison operators such as =, <, >, and <>, you also have the ability to use the SQL wildcards `*` and `%` with the `LIKE` predicate, user-defined values for comparisons on types such as dates, and special built-in functions such as `SUBSTRING`.

```
void PrintResults(DataRow* rows[])
{
  Console::WriteLine();
  String* format = S"{0,-7} {1, -10} {2}";
  Console::WriteLine(format, S"Record", S"First Name", S"Last Name");
  for (int i = 0; i < rows->Count; i++)
  {
    Console::WriteLine(format,
                       __box(i),
                       rows[i]->Item[S"FirstName"],
                       rows[i]->Item[S"LastName"]);
  }
}

void Select()
{
#pragma push_macro("new")
#undef new
  try
  {
    DataRow* rows[];
    // Search on a single column
    rows = employeesTable->Select(S"LastName = 'Archer'");
```

```
    PrintResults(rows);
    // Search on multiple columns
    rows = employeesTable->Select(S"FirstName = 'Tom' "
                                  S"AND LastName = 'Archer'");
    PrintResults(rows);
    // Search using the SQL LIKE predicate
    rows = employeesTable->Select(S"Title LIKE 'Sales*'");
    PrintResults(rows);
    // Search and Sort on two columns
    rows = employeesTable->Select(S"City = 'London'",
                                  S"LastName,FirstName");
    PrintResults(rows);
    // Search and Sort in DESCending order
    rows = employeesTable->Select(S"Region = 'WA'", S"BirthDate
      DESC");
    PrintResults(rows);
    // Search using a user-defined value
    rows = employeesTable->Select(S"BirthDate > #01/01/1964#");
    PrintResults(rows);
    // Search using the SUBSTRING expression
    rows = employeesTable->Select(S"SUBSTRING(HomePhone,2,3) =
      '206'");
    PrintResults(rows);
  }
  catch(Exception* e)
  {
    Console::WriteLine(e->Message);
  }
#pragma pop_macro("new")
}
```

Sorting, Searching, and Filtering Data with `DataView` Objects

When programming "true" .NET applications—as opposed to mixed mode applications— `DataView` objects are used to provide data binding between an application's data and Windows Forms (and Web Forms) controls. However, as this chapter is about using ADO.NET in the context of an MFC application, I think of the `DataView` being associated with the `DataTable` much as a `CView` is associated with a `CDocument`. Therefore, for our purposes `DataView` objects allow you to create multiple concurrent views on the same data so that any change to the data is realized by all the views—depending on their filter.

Table 6-3 `DataViewRowState` Enumeration Values

Member Name	Description
`Added`	Returns new rows.
`CurrentRows`	All current rows—including new, changed, and unchanged rows.
`Detached`	Rows marked as deleted, but not yet removed from the collection.
`ModifiedCurrent`	Each `DataRow` object maintains two versions of its data—an "original" version and a "modified" version. This facilitates such things as the batch operations where you can cancel edits, as seen in "Batch Row Changes." The `ModifiedCurrent` enumeration simply specifies that you want the current versions of modified rows. (Compare with `ModifiedOriginal`.)
`ModifiedOriginal`	Specifies that you want the original values of any modified rows. Compare with `ModifiedCurrent`, which allows you to obtain the current values.
`None`	No rows are returned.
`OriginalRows`	This will return all of the original rows, including those that have been deleted or modified.
`Unchanged`	Only returns rows that have not been modified.

While it's not always obvious, each `DataTable` has a built-in `DataView` object associated with it. This view can be accessed via the `DataTable::DefaultView` property. Additionally, you can create as many `DataView` objects on your `DataTable` as your application needs. Typically these `DataView` objects are created in order to provide a filtered view of your data. Therefore, in this section you'll first see how to create filtered views—both by column value and by row state—and then how to sort and search your view's data. Note that I'll be defining new `DataView` objects in the code snippets in this section, as most times when a `DataView` is used it is because the application needs multiple views on the same data. However, you can apply the same techniques to the `DataTable::DefaultView` if your application has no need for a second view.

Filtering on Row State Information

In the section entitled "Working with Row State Information" you learned about the `DataRow::RowState` property and how various operations affect

its value. In addition to being able to create views that are filtered on column values, you can also create filtered views specific to rows having a given `RowState` value. In fact, implementing this feature can be a very powerful addition to your application. For example, let's say that in your disconnected application you wish to provide an Undo feature for any changes the user has made to the `DataTable`. As you'll soon see, you could easily construct a view and display a dialog with all modified, added, or deleted rows so that the user could choose which ones were to be undone. You could also use the same strategy in presenting a dialog that would allow the user to confirm which data to actually update with the server data source. This is all done simply by instantiating a `DataView` object and setting its `RowStateFilter` property to one of the `DataViewRowState` enumeration values shown in Table 6–3.

The following code was once again run against the Employees table, with the difference being that this time I specified in the data adapter's constructor that I only wanted records with `EmployeeID` values less than 4. (This was done merely to produce a shorter printout from the example code.)

```
void DataViewFilterByRowState()
{
#pragma push_macro("new")
#undef new
  try
  {
    // Add rows
    DataRow* row1 = employeesTable->NewRow();
    row1->Item["FirstName"] = S"Fred";
    row1->Item["LastName"] = S"Flintstone";
    employeesTable->Rows->Add(row1);
    // Edit row
    employeesTable->Rows->Item[0]->Item[S"FirstName"] = S"Test";

     // Delete row
     employeesTable->Rows->Item[1]->Delete();
    PrintDataViewInfo(DataViewRowState::OriginalRows);
    PrintDataViewInfo(DataViewRowState::CurrentRows);
    PrintDataViewInfo(DataViewRowState::Added);
    PrintDataViewInfo(DataViewRowState::Deleted);
    PrintDataViewInfo(DataViewRowState::ModifiedCurrent);
    PrintDataViewInfo(DataViewRowState::ModifiedOriginal);
    PrintDataViewInfo(DataViewRowState::Unchanged);
```

```
  }
  catch(Exception* e)
  {
    Console::WriteLine(e->Message);
  }
#pragma pop_macro("new")
}
```

As you can see, the first thing I did was simply to add a new row, edit an existing row, and then delete a row. The `PrintDataViewInfo` function (shown next) is then called to test how these three changes (four if you include the initial reading of the data) would impact a `DataView` using each of the various `DataViewRowState` enumerations.

```
void PrintDataViewInfo(DataViewRowState rowStateFilter)
{
#pragma push_macro("new")
#undef new
  try
  {
    DataView* view = new DataView(employeesTable);
    view->RowStateFilter = rowStateFilter;
    Console::WriteLine(S"Using RowStateFilter = {0}",
                       __box(view->RowStateFilter));
    Console::WriteLine(S"Row count = {0}", __box(view->Count));
    String* format = S"{0,-3} {1, -10} {2}";
    Console::WriteLine(format, S"Row", S"First Name", S"Last Name");
    for (int i = 0; i < view->Count; i++)
    {
      Console::WriteLine(format,
                         __box(i),
                         view->Item[i]->Item[S"FirstName"],
                         view->Item[i]->Item[S"LastName"]);
    }
    Console::WriteLine();
  }
  catch(Exception* e)
  {
    Console::WriteLine(e->Message);
  }
#pragma pop_macro("new")
}
```

As you can see, the `PrintDataViewInfo` function constructs a `DataView` object based on the Employees table, sets the `RowStateFilter` to the passed `DataViewRowState` value, and then simply enumerates and outputs the records for that newly created view. Here is the output from running these two functions:

```
Using RowStateFilter Added
Row count = 1
Row First Name Last Name
0   Fred       Flintstone

Using RowStateFilter OriginalRows
Row count = 3
Row First Name Last Name
0   Nancy      Davolio
1   Andrew     Fuller
2   Janet      Leverling

Using RowStateFilter Deleted
Row count = 1
Row First Name Last Name
0   Andrew     Fuller

Using RowStateFilter CurrentRows
Row count = 3
Row First Name Last Name
0   Test       Davolio
1   Janet      Leverling
2   Fred       Flintstone

Using RowStateFilter ModifiedOriginal
Row count = 1
Row First Name Last Name
0   Nancy      Davolio

Using RowStateFilter ModifiedCurrent
Row count = 1
Row First Name Last Name
0   Test       Davolio
```

As you can see, with the `DataView` you can retrieve the original rows that were read into the `DataTable`, filter only added or deleted rows, view

all current rows (which is what the `DataTable::DefaultView` does), or even specify that you want to see only modified rows (either the original values or the current values).

Filtering on Column Values

Much as you can use the `DataTable::Select` method to search for rows giving a search criteria, you can also search `DataView` objects via the `DataView::RowFilter` property. The main difference is that the `Select` method returns an array of `DataRow` objects, while the `RowFilter` property simply changes the records that the `DataView` presents when enumerating the row collection or requesting a specific row (via the `Item` property or one of the search methods).

In addition, the `RowFilter` takes a `String` parameter that acts as the search criterion that allows the same expression values as the `Select` method. Therefore, I'll refer you back to the "Searching (and Sorting) on Column Values" section in order to see examples of what types of expressions you can pass to the `RowFilter` property.

Sorting

Along with filtering `DataView` objects, you can also sort them via the `DataView::Sort` property. In order to set this value, simply specify the name of one or more columns separated by commas and include either ASC for (ascending sort) or DESC (for descending sort). Note that if you do not specify ASC or DESC for a given column, then its sort order defaults to ascending.

```
...

DataView* view = new DataView(employeesTable);
// Sort by a single column in ascending order
view->Sort = S"LastName";
// Sort by birthdate in descending order
view->Sort = S"BirthDate DESC";
// Sort by multiple columns in various orders
view->Sort = S"LastName, BirthDate DESC, Title";
```

Working with Typed Datasets

As mentioned earlier in the chapter, typed datasets are not a separate set of ADO.NET classes. Instead, they are derived directly from the standard ADO.NET classes and define extra members that are specific to your database schema. To understand the main benefit of typed datasets, think of how data has been accessed throughout this chapter using the generic `DataRow::Item` property, where column name or index value is specified. Because the `Item` property has no intimate knowledge of the underlying data schema, any mistakes made when using it—such as such as misspelling the column name or using an incompatible data type—aren't realized until runtime. Conversely, typed datasets are generated directly from the database schema and thus produce members that allow for compile-time checking of these common mistakes.

In this section, I'll first illustrate—step-by-step—how to generate a typed dataset for the SQL Server sample Northwind database's Employees table. After that, I'll introduce an MFC SDI application that uses a typed dataset to list all records from the Employees table. Finally, I'll wrap up the section—and chapter—with a listing of the pros and cons of using untyped vs. typed datasets.

Generating a Typed Dataset

Now let's see how to use Visual Studio .NET to generate a typed dataset. As stated at the outset of this chapter, the demo will be using the SQL Server sample Northwind database.

1. Either create a new project with support for Managed Extensions, or open an existing Managed Extensions project.
2. Select the Add New Item option from the Project menu (or from the project's context menu in the Solution Explorer).
3. When the Add New Item dialog appears (Figure 6–5), click the Data folder (category) on the left and then the `DataSet` icon (template) on the right.
4. Enter a name for the header file that will contain the definitions of the typed dataset and any included `DataTable` objects. While this demo will only use the Northwind Employees table, you might want add other `DataTable` definitions later, so name the file `Northwind` and click the Open button.

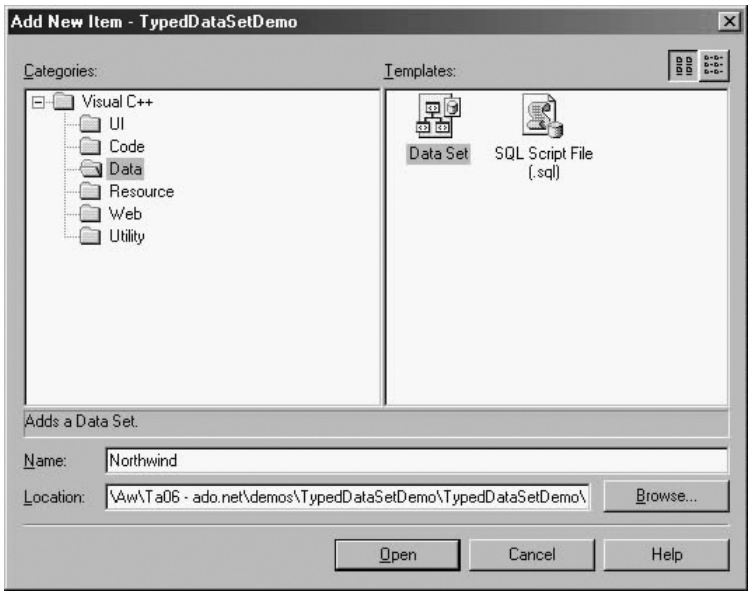

Figure 6–5 Visual Studio .NET provides a template for generating typed datasets.

5. At this point a script will run that will generate the file containing the typed dataset information. *It's important to realize that this script might trigger a response from your anti-virus software.* If that is the case, just select the option to allow the script to run. The Visual Studio .NET Output window will display the progress of the script.
6. When a typed dataset file is generated, an XSD file (XML Schema Definition) file is created to describe the dataset. This file is automatically opened upon the creation of the typed dataset (Figure 6–6) in a window that is sometimes called a *canvas.* Database entities can be added to the dataset by dragging them from the Server Explorer onto the canvas, so at this point, click the canvas's Server Explorer link (also available from the Project menu).
7. If the desired database is on your local machine, simply browse to that database using the Server Explorer tree view. However, if your database is on a remote machine, you'll need to click the Connect to Server button at the top of the Server Explorer dialog. When the Add Server dialog is displayed, simply type the name of the server. (There's no need to add the UNC path information.)

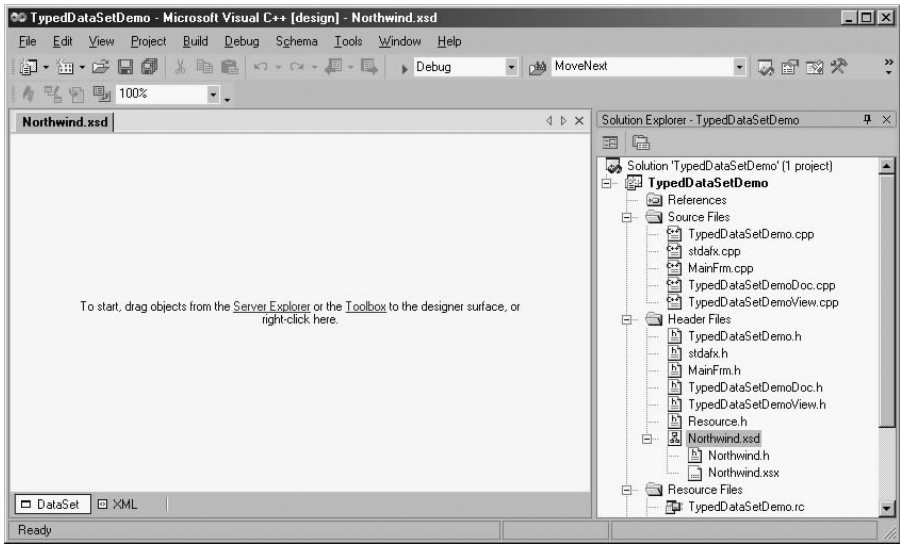

Figure 6–6 A drop-target canvas is used to visually represent a typed dataset.

8. At this point, you should have a database displayed in the Server Explorer—local or remote. Figure 6–7 shows my particular configuration. As you can see, JEANVALJEAN is the name of my programmer machine, and it has no SQL Server databases. Therefore, I added (connected to) a server named FANTINE to my Server Explorer so that I could access its SQL Server databases.
9. While the icons next to the various databases include an "x", once you expand the item the Server Explorer connects to the database and retrieves the selected database's information—such as its Tables, Views, Stored Procedures, and so on. Here you would simply drag the desired database entity to the dataset palette. For purposes of this demo—and so we can make an "apples and apples" comparison between the untyped datasets you've been using throughout the chapter and a typed dataset—drag the Employees table from the Northwind database to the typed dataset canvas.
10. If you open the `Northwind.h` file, you won't see any Employee `DataTable` information. That information isn't generated until you build the project. Building the project now will result in Visual Studio .NET generating the expected classes and members in the `Northwind.h` file for the Employees table.

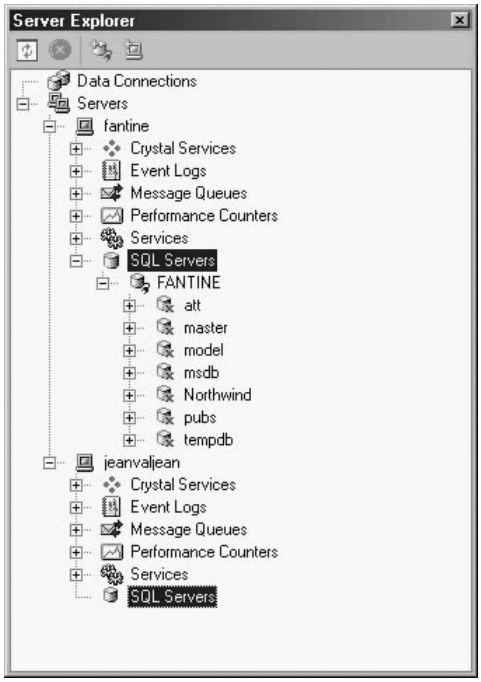

Figure 6–7 The Server Explorer allows you to drag database entities—such as tables, views, and stored procedures—onto the typed dataset view.

A tremendous amount of code is actually generated for a typed dataset, so we'll just briefly look at some of it—but enough so that you get an idea of what all has been done for you. When you open the `Northwind.h` file, the first thing you notice is that the file is free-standing in that the appropriate `#using` and `using` namespace directives have been inserted. Also note that the typed dataset classes have been defined within a namespace of the same name as the current project. That means you'll have to either qualify all uses of the types defined in this file or insert a `using` namespace directive into any of your typed dataset client code.

The first class that you encounter will be the `DataSet`-derived Northwind class. This class also defines nested classes for the Employees table that include a `DataTable`-derived class, a `DataRow`-derived class, and a delegate for subscribing to row-change events.

```
public __gc class Northwind : public System::Data::DataSet {
  public : __gc class EmployeesDataTable;
```

```
  public : __gc class EmployeesRow;
  public : __gc class EmployeesRowChangeEvent;
  ...
```

You'll note that the file does not include any connection-specific information as you might expect. This is so that the typed dataset can be used against any connection. Therefore, the typed dataset only refers to the data entity's schema—not to where the underlying data store is located or how to connect to it. Therefore, the way you would use these types is to connect and build a data adapter just as you would with an untyped dataset, and then use the typed `DataSet`-derived and `DataTable`-derived classes as follows.

```
#include "Northwind.h"
using namespace System::Data::SqlClient;
using namespace TypedDataSetDemo;

...

SqlConnection* conn =
  new SqlConnection(S"Server=FANTINE;"
                    S"Database=Northwind;"
                    S"Integrated Security=true;");
SqlDataAdapter* adapter =
  new SqlDataAdapter(S"SELECT * FROM Employees", conn);
SqlCommandBuilder* commandBuilder = new SqlCommandBuilder(adapter);
conn->Open();
Northwind* northwind = new Northwind();
adapter->Fill(northwind, S"Employees");
conn->Close(); // No longer needed
Northwind::EmployeesDataTable* employees = northwind->Employees;

...
```

Let's look at this code—especially in comparison to the connection code used previously in this chapter using untyped datasets. Obviously, I need to include the `Northwind.h` file, as it defines the typed dataset types. After that, I need to specify the `System::Data::SqlClient` namespace, as the `Northwind.h` file is using generic base classes and the `SqlClient` namespace is specific to SQL Server ADO.NET types. Finally, the `TypedDataSetDemo` namespace is referenced so that we don't have to qualify the references to the various typed dataset types.

After the standard connection object creation and instantiation of a data adapter object, I declare an instance of the `DataSet`-derived Northwind class. From there, I call the adapter's `Fill` method. However, note that I've changed my table name from the `"AllEmployees"` value that I've used throughout the chapter to `"Employees"`. While seemingly trivial, I'm actually forced to do this because the `Northwind::Employees` property is hard-coded to return an `EmployeesDataTable` object with a name of `"Employees"`.

One last thing that I'll point out before you see a demo illustrating how to use a typed dataset in an MFC application is that no `DataRowCollection`-derived class is generated for us. This is because the `EmployeesDataTable` implements the `IEnumerable` interface and acts as a collection for its rows. The various collection classes, the `IEnumerable` interface, and even creating your own enumerable classes are covered in Chapter 11. The `EmployeesDataTable` also implements the `Count` and `Item` properties so that an `EmployeesDataTable` object can be iterated like an array.

```
public :
[System::Diagnostics::DebuggerStepThrough]
__gc class EmployeesDataTable :
  public System::Data::DataTable,
  public System::Collections::IEnumerable
{

...

__property System::Int32 get_Count();

...

public:
__property TypedDataSetDemo::Northwind::EmployeesRow* get_Item(
  System::Int32 index

);
```

Because these properties have been implemented, the `EmployeesDataTable` can be enumerated as simply as this:

```
...

// For every employee record
for (int i = 0; i < employees->Count; i++)
```

```
{
  int id = *dynamic_cast<__box int*>(employees->Item[i]->EmployeeID);
  String* firstName = employees->Item[i]->FirstName;
  String* lastName = employees->Item[i]->LastName;
}
```

As you can see, each row is retrieved via the `Item` property, and, as the row is a `Northwind::EmployeesRow`, there are properties with the same names and types as the actual Employees table:

```
public : [System::Diagnostics::DebuggerStepThrough]
__gc class EmployeesRow : public System::Data::DataRow {

...

public: __property System::Int32 get_EmployeeID();
public: __property  void set_EmployeeID(System::Int32 value);
public: __property System::String*  get_LastName();
public: __property  void set_LastName(System::String*_u32 ?value);
public: __property System::String*  get_FirstName();
public: __property  void set_FirstName(System::String*  value);

...
```

Using Typed Datasets

Now let's see how to use typed dataset types in an MFC demo. We'll keep the application simple, as it will connect to the Employees table, read all the employee records, and display them in a list view. However, combining what you'll see here with what you've learned throughout the chapter should make it easy for you to use typed datasets for all your dataset needs should you decide to go that route.

1. To get started, create and MFC SDI application called `Typed DataSetDemo`, setting the view base class to `CListView` and updating the Project settings to support Managed Extensions.
2. As explained in the previous section, create a typed dataset called `Northwind` and add to it the Employees table.
3. Add the following directives to the `stdafx.h` file:

```
#using <mscorlib.dll>
#using <system.dll>
```

```
#using <system.windows.forms.dll>
using namespace System;
using namespace System::Windows::Forms;
#undef MessageBox
```

4. Open the `TypedDataSetDemoView.h` file and add the following directives just before the `CTypedDataSetDemoView` class declaration.

```
#include "Northwind.h"
using namespace System::Data::SqlClient;
using namespace TypedDataSetDemo;
```

5. Declare the following ADO.NET objects in the `CTypedDataSetDemoView` class.

```
class CTypedDataSetDemoView : public CListView
{
...
protected:
  gcroot<Northwind*>northwind;
  gcroot<SqlDataAdapter*>adapter;
  gcroot<Northwind::EmployeesDataTable*>employees;
  gcroot<SqlCommandBuilder*>commandBuilder;
...
```

6. Add the following code to the view's `OnInitialUpdate` function to initialize the list view.

```
void CTypedDataSetDemoView::OnInitialUpdate()
{
   CListView::OnInitialUpdate();
  // Add columns to listview
  CListCtrl& lst = GetListCtrl();
  lst.InsertColumn(0, _T("ID"));
  lst.InsertColumn(1, _T("First Name"));
  lst.InsertColumn(2, _T("Last Name"));
...
```

7. In the view class's `PreCreateWindow` function, set the view window's style to "report":

```
BOOL CTypedDataSetDemoView::PreCreateWindow(CREATESTRUCT& cs)
{
  cs.style |= LVS_REPORT;
  return CListView::PreCreateWindow(cs);
}
```

8. Insert the following code just before the end of the `OnInitialUpdate` function.

```
#pragma push_macro("new")
#undef new
try
  {
    SqlConnection* conn =
      new SqlConnection(S"Server=localhost;"
                        S"Database=Northwind;"
                        S"Integrated Security=true;");
    adapter = new SqlDataAdapter(S"SELECT * FROM Employees",
                                 conn);
    commandBuilder = new SqlCommandBuilder(adapter);
    conn->Open();
    northwind = new Northwind();
    adapter->Fill(northwind, S"AllEmployees");
    conn->Close(); // No longer needed
    employees = northwind->Employees;
    ReadAllEmployees();
  }
  catch(Exception* e)
  {
    MessageBox::Show(e->Message,
                     S".NET Exception Thrown",
                     MessageBoxButtons::OK,
                     MessageBoxIcon::Error);
  }
#pragma pop_macro("new")
```

9. Finally, implement the `ReadAllEmployees` function as follows:

```
void CTypedDataSetDemoView::ReadAllEmployees()
{
  try
  {
    CWaitCursor wc;
    CListCtrl& lst = GetListCtrl(); lst.DeleteAllItems();
    for (int i = 0; i < employees->Count; i++)
    {
      int idx = lst.InsertItem(i,
        (CString)__box(employees->Item[i]->EmployeeID)->
          ToString());
      lst.SetItemText(idx, 1,
        (CString)employees->Item[i]->FirstName);
```

```
        lst.SetItemText(idx, 2,
          (CString)employees->Item[i]->LastName);
      }
    }
    catch(Exception* e)
    {
      MessageBox::Show(e->Message, S".NET Exception Thrown",
                       MessageBoxButtons::OK,
                       MessageBoxIcon::Error);
    }
  }
```

Building and running the application will result in something similar to what you see in Figure 6–8. The code in this section was really not much different than what you've seen throughout this chapter, with the principal difference being that the use of typed dataset types enabled you to have the compiler check for common errors as opposed to having them realized at runtime. This segues nicely into the next part of this chapter—weighing the pros and cons of using typed datasets.

Weighing the Pros and Cons of Typed Datasets

Covering the entirety of typed datasets would take an entire chapter by itself. However, you've learned enough about typed datasets in this section to realize some of the positive points regarding using them in your ADO.NET code. Let's now briefly look at some of the advantages and disadvantages of using typed datasets (vs. using untyped datasets).

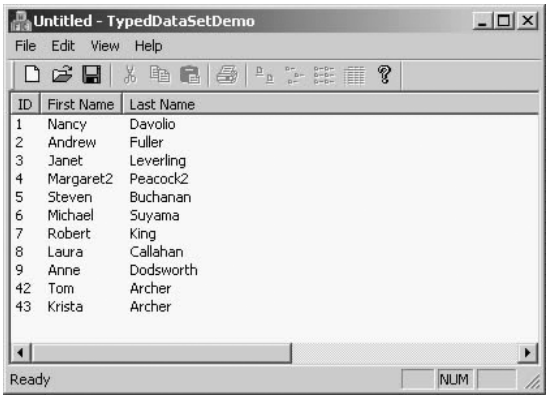

Figure 6–8 Example of running the `TypedDataSetDemo` application

Advantages of typed datasets:

- **Compile-time type checking—**Reduces runtime errors by having members based on the data's actual schema as opposed to untyped datasets, where you call a generic function and can pass an object of any type.
- **Schema-specific members—**Typed datasets define properties for getting and setting values where the property name is the same as the underlying column name. They also define properties for determining if the column is null and methods for searching the table via primary key(s).
- **Data binding support in VS.NET—**Only useful with Windows Forms applications, but bears mentioning if you plan on doing development based entirely on .NET in addition to the mixed-mode programming that is the focus of this book.
- **Intellisense support—**When using untyped datasets, you have to know beforehand the names of the columns and the types that the respective columns work with. With typed datasets, as soon as you enter the name of the type, Intellisense displays its members, thereby saving you development and debugging time.

Disadvantages of typed datasets:

- **Versioning**—Typed datasets can actually increase development time in situations where your schema changes, because you'll need to update the typed dataset information manually. This is obviously the same problem we've battled for years with `CRecordset` classes—having to modify them manually when the underlying schema changes.
- **Tightly coupled**—In its current design, typed datasets are difficult to extend and can't be modified (as they're auto-generated each time the project is built). In addition, they force a tight coupling of client to data access code, which might not be best for all situations.

As mentioned earlier in the chapter, I use untyped datasets in code snippets and demos, as this provides for shorter, to-the-point code listings. However, I personally choose to use typed datasets in my production code, as once I'm to the implementation stage of a project, I'm finished with any major changes to the database schema, and with the introduction of dataset

annotations, the only two real disadvantages for me are moot. (Annotations are extensions to the XSD format used to define typed datasets that allow some changes such as the renaming of classes and properties as well as defining how to deal with null values.)

Summary

The chapter began with a section illustrating the basic terminology and classes specific to ADO.NET and disconnected data as well as the basics of constructing those objects and connecting to a data source. The next section illustrated how to perform basic database operations (creating, reading, updating, and deleting records). Included in that section were tips on handling the common issues of dealing with disconnected data and auto-increment primary keys, and how to specify primary key information for untyped datasets. The section ended with a fully functional MFC maintenance application using the ADO.NET against a sample database. Once that section was completed, I then illustrated several techniques for reading and writing data in batch in order to keep round trips to the data server to a minimum and dramatically speed up insert operations. As one of the key benefits of an IMDB is its support of the ability to search, sort, and filter data once it's been retrieved, the next section illustrated how to accomplish these tasks as well as how to create multiple filtered views on a single dataset. Finally, the chapter ended with a section on generating strongly typed datasets, using them in an MFC application, and the pros and cons of using untyped vs. typed datasets.

While disconnected data can make applications more complex, it also dramatically improves the versatility and design of client/server and *n*-tier systems. For example, disconnected data means that systems with thousands of remote clients do not have to worry about how the server is going to attempt to handle so many concurrent connections to the database. In addition, because you can design your system so that the majority of the database operations are done in-memory before updating the database server, you can greatly reduce the number of network round trips. Using disconnected data also enables you to fill a `DataTable` with data taken from virtually any data source (or even application-generated) and process it with the same routines, regardless of the data's origin. In addition, the decoupled architecture of the `DataSet`- and `DataAdapter`-derived classes makes it possible to read data from one source and send updates to another

source. These are just a few of the possible scenarios where disconnected data can be a boon for software developers.

Please note that I'm not saying disconnected data is a silver bullet by any means. Every solution to a problem has its own drawbacks. In fact, using disconnected data is flat-out a very bad idea in certain circumstances. Specifically, just as with any other solution that introduces a certain level of complexity to a system, you must weigh the added complexity against the derived benefits. However, for systems—especially *n*-tier solutions—where both database connections and round trips must kept to a minimum, disconnected data via the ADO.NET class can significantly improve the overall design and execution of the system.

Finally, there's a lot more even to the parts of ADO.NET that support disconnected data than I could possibly convey in a single chapter. However, hopefully you've learned enough in this chapter to decide for yourself if and when ADO.NET and disconnected data are right for you. Now that you know these basics, in the next chapter I'll delve into the more advanced areas of ADO.NET such as merging `DataSet` objects, reading binary data (especially BLOBs), dealing with concurrency in a disconnected environment, and so on.