18

Database: LINQ – Microsoft Language-Integrated Query

18.6 Microsoft Language Integrated Query (LINQ)

- The next several sections introduce C#'s LINQ (Language Integrated Query) capabilities.
- LINQ allows you to write query expressions, similar to SQL queries, that retrieve information from a wide variety of data sources, not just databases.
- We use LINQ to Objects in this section to query arrays and Lists, selecting elements that satisfy a set of conditions—this is known as filtering.

18.6.1 Querying an Array of int Values Using LINQ

- Repetition statements that filter arrays focus on the process of getting the results—iterating through the elements and checking whether they satisfy the desired criteria.
- LINQ specifies the conditions that selected elements must satisfy. This is known as **declarative programming**—as opposed to **imperative programming** (which we've been doing so far) in which you specify the actual steps to perform a task.

18.6.1 Querying an Array of int Values Using LINQ (Cont.)

- The next several statements assume that the integer array

```
int[] values = { 2, 9, 5, 0, 3, 7, 1, 4, 8, 5 };
```

is declared. The query

```
var filtered =
    from value in values
    where value > 4
    select value;
```

specifies that the results should consist of all the `int`s in the `values` array that are greater than 4 (i.e., 9, 5, 7, 8 and 5). It *does not* specify *how* those results are obtained—the C# compiler generates all the necessary code automatically, which is one of the great strengths of LINQ.

18.6.1 Querying an Array of int Values Using LINQ (Cont.)

The from Clause and Implicitly Typed Local Variables

- A LINQ query begins with a `from` clause, which specifies a range variable (`value`) and the data source to query (`values`).
- The range variable represents each item in the data source (one at a time), much like the control variable in a `foreach` statement. We do not specify the range variable's type.
- Implicitly typed local-variables enable the compiler to *infer* a local variable's type based on the context in which it's used.

18.6.1 Querying an Array of int Values Using LINQ (Cont.)

The var Keyword and Implicitly Typed Local Variables

- You can also declare a local variable and let the compiler infer the variable's type based on the variable's initializer. To do so, the **var keyword** is used in place of the variable's type when declaring the variable. Consider the declaration

```
var x = 7;
```

- Here, the compiler *infers* that the variable x should be of type `int`, because the compiler assumes that whole-number values, like 7, are of type int.
- Similarly, in the declaration

```
var y = -123.45;
```

the compiler infers that y should be of type double, because the compiler assumes that floating-point number values, like `-123.45`, are of type double

18.6.1 Querying an Array of int Values Using LINQ (Cont.)

The where Clause

- If the condition in the `where` clause evaluates to true, the element is *selected*—i.e., it's included in the results.
- An expression that takes an element of a collection and returns `true` or `false` by testing a condition on that element is known as a predicate.

The `select` Clause

- For each item in the data source, the `select` clause determines what value appears in the results. A LINQ query typically ends with a `select` clause.

18.6.1 Querying an Array of int Values Using LINQ (Cont.)

Iterating Through the Results of the LINQ Query

- The foreach statement

```
foreach ( var element in filtered )
   Console.Write( " {0}", element );
```

displays the query results.

- A foreach statement can iterate through the contents of an array, collection or the results of a LINQ query, allowing you to process each element in the array, collection or query.
- The preceding foreach statement iterates over the query result filtered, displaying each of its items.

18.6.1 Querying an Array of int Values Using LINQ (Cont.)

The orderby Clause

- The orderby clause sorts the query results in ascending order. The query

```
var sorted =
   from value in values
   orderby value
   select value;
```

sorts the integers in array values into ascending order and assigns the results to variable sorted.

To sort in descending order, use descending in the orderby clause, as in

```
orderby value descending
```

- An ascending modifier also exists but isn't normally used, because it's the default.

18.6.1 Querying an Array of int Values Using LINQ (Cont.)

- The following two queries generate the same results, but in different ways:

```
var sortFilteredResults =
    from value in filtered
    orderby value descending
    select value;
var sortAndFilter =
    from value in values
    where value > 4
    orderby value descending
    select value;
```

18.6.1 Querying an Array of int Values Using LINQ (Cont.)

- The first uses LINQ to sort the results of the `filtered` query presented earlier in this section.
- The second query uses both the where and `orderby` clauses.

An Aside: Interface IEnumerable<T>

- `foreach` iterates over any so-called `IEnumerable<T>` object, which just happens to be what a LINQ query returns. `IEnumerable<T>` is an interface that describes the functionality of any object that can be iterated over and thus offers methods to access each element.

18.6.2 Querying an Array of Employee Objects Using LINQ

- LINQ can be used with most data types, including strings and user-defined classes.
- The query result's `Any` method returns `true` if there's at least one element, and `false` if there are no elements.
- The query result's `First` method returns the first element in the result.
- You should check that the query result is not empty before calling `First`.
- LINQ defines many more extension methods, such as `Count`, which returns the number of elements in the results.
- The `Distinct` extension method removes duplicate elements, causing all elements in the result to be unique.

18.6.2 Querying an Array of Employee Objects Using LINQ

- The syntax

```
new { e.FirstName, Last = e.LastName }
```

creates a new object of an anonymous type (a type with no name), which the compiler generates for you based on the properties listed in the curly braces ({}).

- In this case, the anonymous type consists of properties for the first and last names of the selected `Employee`.
- The `LastName` property is assigned to the property `Last` in the `select` clause.
- This is an example of a projection—it performs a transformation on the data.

```
1  // Fig. 18.23: Employee.cs
2  // Employee class with FirstName, LastName and MonthlySalary properties.
3  public class Employee
4  {
5     private decimal monthlySalaryValue; // monthly salary of employee
6
7     // auto-implemented property FirstName
8     public string FirstName { get; set; }
9
10    // auto-implemented property LastName
11    public string LastName { get; set; }
12
13    // constructor initializes first name, last name and monthly salary
14    public Employee( string first, string last, decimal salary )
15    {
16       FirstName = first;
17       LastName = last;
18       MonthlySalary = salary;
19    } // end constructor
20
```

Fig. 18.23 | Employee class. (Part 1 of 2.)

```
21      // property that gets and sets the employee's monthly salary
22      public decimal MonthlySalary
23      {
24         get
25         {
26            return monthlySalaryValue;
27         } // end get
28         set
29         {
30            if ( value >= 0M ) // if salary is nonnegative
31            {
32               monthlySalaryValue = value;
33            } // end if
34         } // end set
35      } // end property MonthlySalary
36
37      // return a string containing the employee's information
38      public override string ToString()
39      {
40         return string.Format( "{0,-10} {1,-10} {2,10:C}",
41            FirstName, LastName, MonthlySalary );
42      } // end method ToString
43   } // end class Employee
```

Fig. 18.23 | Employee class. (Part 2 of 2.)

```
1  // Fig. 18.24: LINQWithArrayOfObjects.cs
2  // LINQ to Objects using an array of Employee objects.
3  using System;
4  using System.Linq;
5
6  public class LINQWithArrayOfObjects
7  {
8     public static void Main( string[] args )
9     {
10       // initialize array of employees
11       Employee[] employees = {
12          new Employee( "Jason", "Red", 5000M ),
13          new Employee( "Ashley", "Green", 7600M ),
14          new Employee( "Matthew", "Indigo", 3587.5M ),
15          new Employee( "James", "Indigo", 4700.77M ),
16          new Employee( "Luke", "Indigo", 6200M ),
17          new Employee( "Jason", "Blue", 3200M ),
18          new Employee( "Wendy", "Brown", 4236.4M ) }; // end init list
19
20       // display all employees
21       Console.WriteLine( "Original array:" );
22       foreach ( var element in employees )
23          Console.WriteLine( element );
```

Fig. 18.24 | LINQ to Objects using an array of `Employee` objects. (Part 1 of 6.)

```
24
25          // filter a range of salaries using && in a LINQ query
26          var between4K6K =
27             from e in employees
28             where e.MonthlySalary >= 4000M && e.MonthlySalary <= 6000M
29             select e;
30
31          // display employees making between 4000 and 6000 per month
32          Console.WriteLine( string.Format(
33             "\nEmployees earning in the range {0:C}-{1:C} per month:",
34             4000, 6000 ) );
35          foreach ( var element in between4K6K )
36             Console.WriteLine( element );
37
38          // order the employees by last name, then first name with LINQ
39          var nameSorted =
40             from e in employees
41             orderby e.LastName, e.FirstName
42             select e;
43
44          // header
45          Console.WriteLine( "\nFirst employee when sorted by name:" );
46
```

Fig. 18.24 | LINQ to Objects using an array of `Employee` objects. (Part 2 of 6.)

```
47          // attempt to display the first result of the above LINQ query
48          if ( nameSorted.Any() )
49             Console.WriteLine( nameSorted.First() );
50          else
51             Console.WriteLine( "not found" );
52
53          // use LINQ to select employee last names
54          var lastNames =
55             from e in employees
56             select e.LastName;
57
58          // use method Distinct to select unique last names
59          Console.WriteLine( "\nUnique employee last names:" );
60          foreach ( var element in lastNames.Distinct() )
61             Console.WriteLine( element );
62
63          // use LINQ to select first and last names
64          var names =
65             from e in employees
66             select new { e.FirstName, Last = e.LastName };
67
68          // display full names
69          Console.WriteLine( "\nNames only:" );
```

Fig. 18.24 | LINQ to Objects using an array of `Employee` objects. (Part 3 of 6.)

```
70          foreach ( var element in names )
71             Console.WriteLine( element );
72
73          Console.WriteLine();
74       } // end Main
75    } // end class LINQWithArrayOfObjects
```

Fig. 18.24 | LINQ to Objects using an array of `Employee` objects. (Part 4 of 6.)

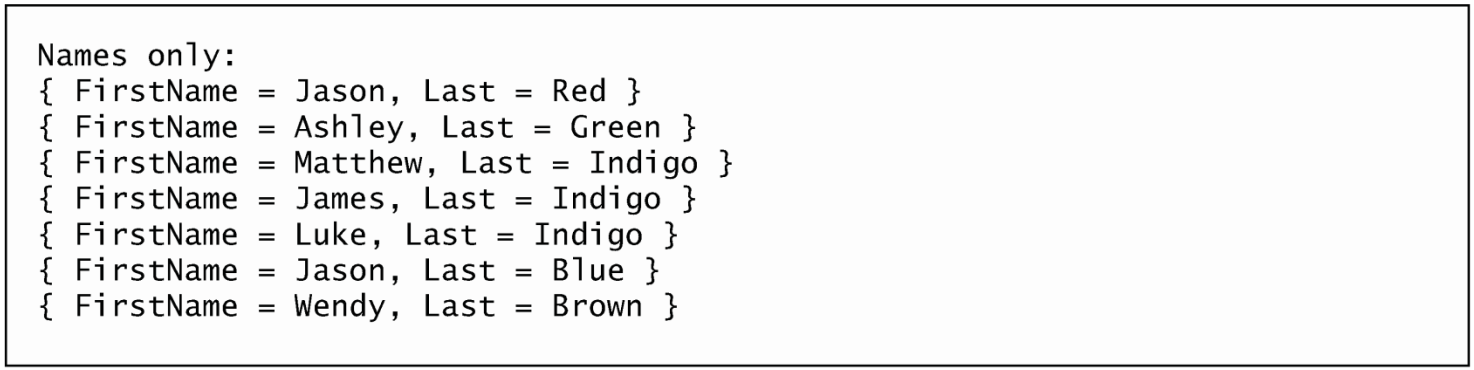
```
Original array:
Jason       Red          $5,000.00
Ashley      Green        $7,600.00
Matthew     Indigo       $3,587.50
James       Indigo       $4,700.77
Luke        Indigo       $6,200.00
Jason       Blue         $3,200.00
Wendy       Brown        $4,236.40

Employees earning in the range $4,000.00-$6,000.00 per month:
Jason       Red          $5,000.00
James       Indigo       $4,700.77
Wendy       Brown        $4,236.40

First employee when sorted by name:
Jason       Blue         $3,200.00

Unique employee last names:
Red
Green
Indigo
Blue
Brown
```

Fig. 18.24 | LINQ to Objects using an array of `Employee` objects. (Part 5 of 6.)



```
Names only:
{ FirstName = Jason, Last = Red }
{ FirstName = Ashley, Last = Green }
{ FirstName = Matthew, Last = Indigo }
{ FirstName = James, Last = Indigo }
{ FirstName = Luke, Last = Indigo }
{ FirstName = Jason, Last = Blue }
{ FirstName = Wendy, Last = Brown }
```

Fig. 18.24 | LINQ to Objects using an array of `Employee` objects. (Part 6 of 6.)

18.6.3 Querying a Generic Collection Using LINQ

- You can use LINQ to Objects to query Lists just as arrays.
- LINQ's `let` clause allows you to create a new range variable.
  - Useful if you need to store a temporary result for use later in the LINQ query.
  - Typically, `let` declares a new range variable to which you assign the result of an expression that operates on the query's original range variable.
- `string` method `ToUpper` converts each item to uppercase, then stores the result in the new range variable `uppercaseString`.
- The `where` clause uses string method `StartsWith` to determine whether `uppercaseString` starts with the character `"R"`.

18.6.3 Querying a Generic Collection Using LINQ (Cont.)

- deferred execution
  - the query executes only when you access the results—such as iterating over them or using the `Count` method—not when you define the query.
  - Allows you to create a query once and execute it many times.
  - Any changes to the data source are reflected in the results each time the query executes.
- C# has a feature called collection initializers, which provide a convenient syntax (similar to array initializers) for initializing a collection.

```
1  // Fig. 18.25: LINQWithListCollection.cs
2  // LINQ to Objects using a List< string >.
3  using System;
4  using System.Linq;
5  using System.Collections.Generic;
6
7  public class LINQWithListCollection
8  {
9     public static void Main( string[] args )
10    {
11       // populate a List of strings
12       List< string > items = new List< string >();
13       items.Add( "aQua" ); // add "aQua" to the end of the List
14       items.Add( "RusT" ); // add "RusT" to the end of the List
15       items.Add( "yElLow" ); // add "yElLow" to the end of the List
16       items.Add( "rEd" ); // add "rEd" to the end of the List
17
18       // convert all strings to uppercase; select those starting with "R"
19       var startsWithR =
20          from item in items
21          let uppercaseString = item.ToUpper()
22          where uppercaseString.StartsWith( "R" )
23          orderby uppercaseString
24          select uppercaseString;
```

Fig. 18.25 | LINQ to Objects using a `List<string>`. (Part 1 of 2.)

```
25
26          // display query results
27          foreach ( var item in startsWithR )
28             Console.Write( "{0} ", item );
29
30          Console.WriteLine(); // output end of line
31
32          items.Add( "rUbY" ); // add "rUbY" to the end of the List
33          items.Add( "SaFfRon" ); // add "SaFfRon" to the end of the List
34
35          // display updated query results
36          foreach ( var item in startsWithR )
37             Console.Write( "{0} ", item );
38
39          Console.WriteLine(); // output end of line
40       } // end Main
41    } // end class LINQWithListCollection
```

```
RED RUST
RED RUBY RUST
```

Fig. 18.25 | LINQ to Objects using a `List<string>`. (Part 2 of 2.)

18.7 LINQ to SQL

- LINQ to SQL enables you to access data in *SQL Server databases* using the same LINQ syntax introduced in the previous section.
- You interact with the database via classes that are automatically generated from the database schema by the IDE's LINQ to SQL Designer.

18.7 LINQ to SQL (Cont.)

- For each table in the database, the IDE creates two classes:
  - A class that represents a row of the table. LINQ to SQL creates objects of this class—called row objects—to store the data from individual rows of the table.
  - A class that represents the table: LINQ to SQL creates an object of this class to store a collection of row objects that correspond to all of the rows in the table.

18.7 LINQ to SQL (Cont.)

- Relationships between tables are also taken into account in the generated classes:
  - In a row object's class, an additional property is created for each foreign key.
  - In the class for a row object, an additional property is created for the collection of row objects with foreign-keys that reference the row object's primary key.

18.7 LINQ to SQL (Cont.)

IQueryable Interface

- LINQ to SQL works through the `IQueryable` interface, which inherits from the `IEnumerable` interface.
- When a LINQ to SQL query on an `IQueryable` object executes against the database, the results are loaded into objects of the corresponding LINQ to SQL classes for convenient access in your code.

18.8 Querying a Database with LINQ

- The IDE provides *visual programming* tools and *wizards* that simplify accessing data in applications. These tools
  - establish database connections
  - create the objects necessary to view and manipulate the data through Windows Forms GUI controls—a technique known as data binding.

18.8 Querying a Database with LINQ (Cont.)

- All of the controls in this GUI are automatically generated when we drag a data source that represents the `Authors` table onto the `Form` in Design view.
- The `BindingNavigator` is a collection of controls that allow you to navigate through the records in the `DataGridView` that fills the rest of the window.
- The `BindingNavigator` controls also allow you to add records, delete records and save your changes to the database.

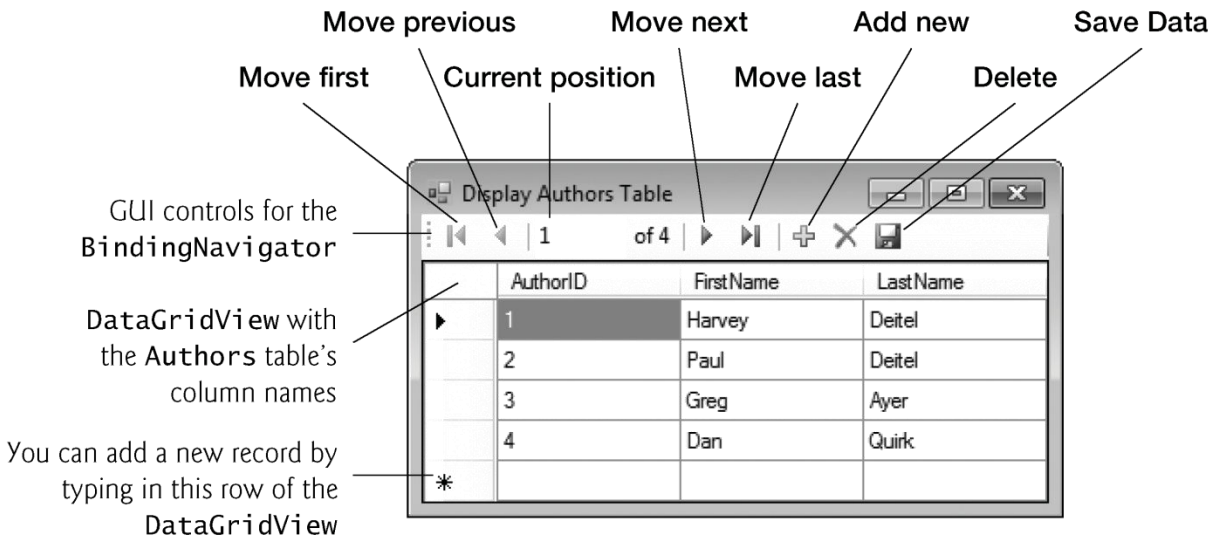


Fig. 18.26 | GUI for the **Display Authors Table** application.

18.8.1 Creating LINQ to SQL Classes

Step 1: Creating the Project

- Create a new Windows Forms Application named `DisplayTable`.
- Change the name of the source file to `DisplayAuthorsTable.cs`.
- The IDE updates the Form's class name to match the source file.
- Set the `Form`'s `Text` property to Display Authors Table.

18.8.1 Creating LINQ to SQL Classes (Cont.)

Step 2: Adding a Database to the Project and Connecting to the Database

- To interact with a database, you must create a connection to the database.
  - In Visual C# 2010 Express, select View > Other Windows > Database Explorer to display the Database Explorer window. If you're using a full version of Visual Studio, select View > Server Explorer to display the Server Explorer.
  - From this point forward, we'll refer to the Database Explorer. If you have a full version of Visual Studio, substitute Server Explorer for Database Explorer in the steps.

18.8.1 Creating LINQ to SQL Classes (Cont.)

- Click the Connect to Database icon at the top of the Database Explorer.
  - If the Choose Data Source dialog appears, select Microsoft SQL Server Database File from the Data source: list.
  - If you check the Always use this selection CheckBox, the IDE will use this type of database file by default when you connect to databases in the future.
  - Click Continue to display the Add Connection dialog.

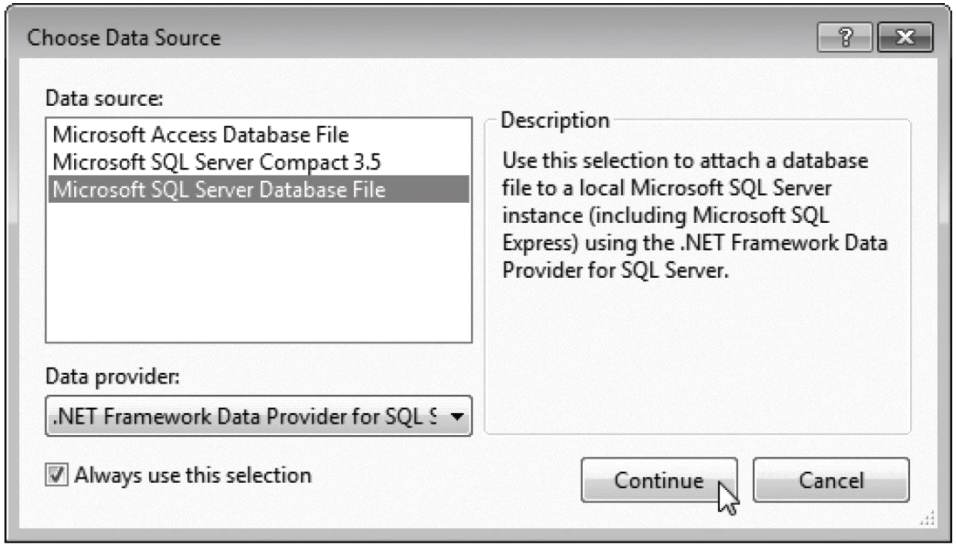


Fig. 18.27 | Choose Data Source dialog.

18.8.1 Creating LINQ to SQL Classes (Cont.)

- In the Add Connection dialog, the Data source: `TextBox` reflects your selection from the Choose Data Source dialog.
  - You can click the Change... `Button` to select a different type of database.
  - Next, click Browse... to locate and select the `Books.mdf` file in the Databases directory included with this chapter's examples.
  - You can click Test Connection to verify that the IDE can connect to the database through SQL Server Express. Click OK to create the connection.

Error-Prevention Tip 18.1

Ensure that no other program is using the database file before you attempt to add it to the project. Connecting to the database requires exclusive access.

Fig. 18.28 | Add Connection dialog.

18.8.1 Creating LINQ to SQL Classes (Cont.)

Step 3: Generating the LINQ to SQL classes

- Right click the project name in the Solution Explorer and select Add > New Item… to display the Add New Item dialog.
- Select the LINQ to SQL Classes template, name the new item `Books.dbml` and click the Add button. The Object Relational -Designer window will appear.
- You can also double click the Books.dbml file in the `Solution Explorer` to open the Object Relational Designer.

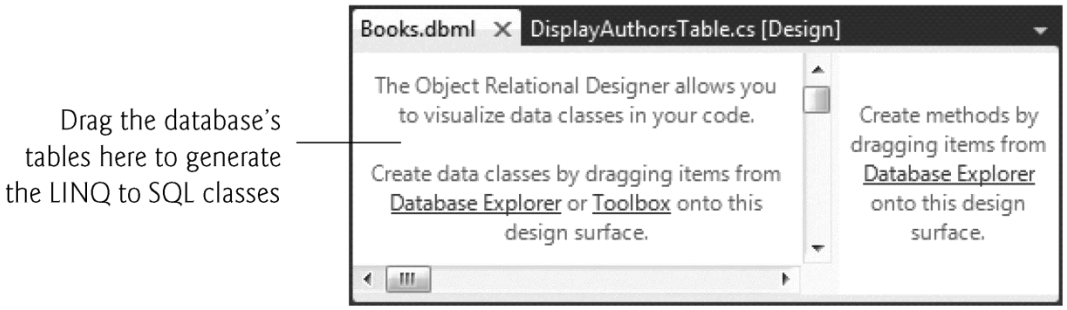


Fig. 18.29 | **Object Relational Designer** window.

18.8.1 Creating LINQ to SQL Classes (Cont.)

- Expand the `Books.mdf` database node in the Database Explorer, then expand the Tables node.
- Drag the `Authors`, `Titles` and `AuthorISBN` tables onto the Object Relational Designer.
- The IDE prompts whether you want to copy the database to the project directory. Select Yes.
- Save the `Books.dbml` file.

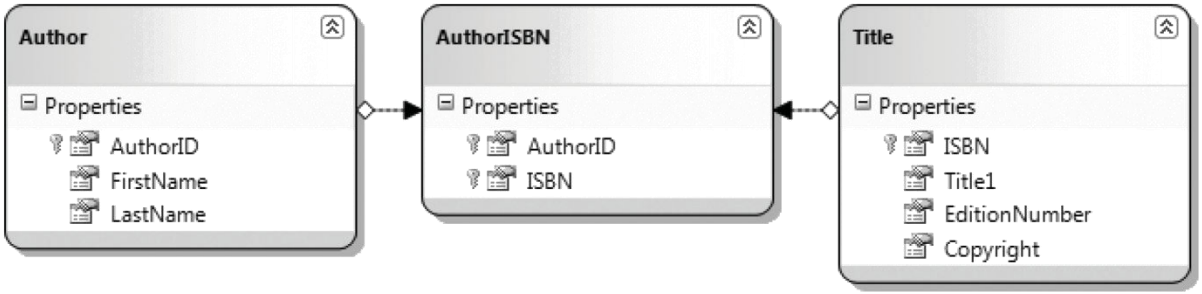


Fig. 18.30 | **Object Relational Designer** window showing the selected tables from the `Books` database and their relationships.

Error-Prevention Tip 18.2

Be sure to save the file in the *Object Relational Designer* before trying to use the LINQ to SQL classes in code. The IDE does not generate the classes until you save the file.

18.8.2 Data Bindings Between Controls and the LINQ to SQL Classes

18.8.2 Data Bindings Between Controls and the LINQ to SQL Classes

Step 1: Adding the Author LINQ to SQL Class as a Data Source

- Select Data > Add New Data Source… to display the Data Source Configuration Wizard.
- The LINQ to SQL classes are used to create objects representing the tables in the database, so we'll use an Object data source.
- In the dialog, select Object and click Next >. Expand the tree view as shown in and ensure that Author is checked. An object of this class will be used as the data source.
- Click Finish.

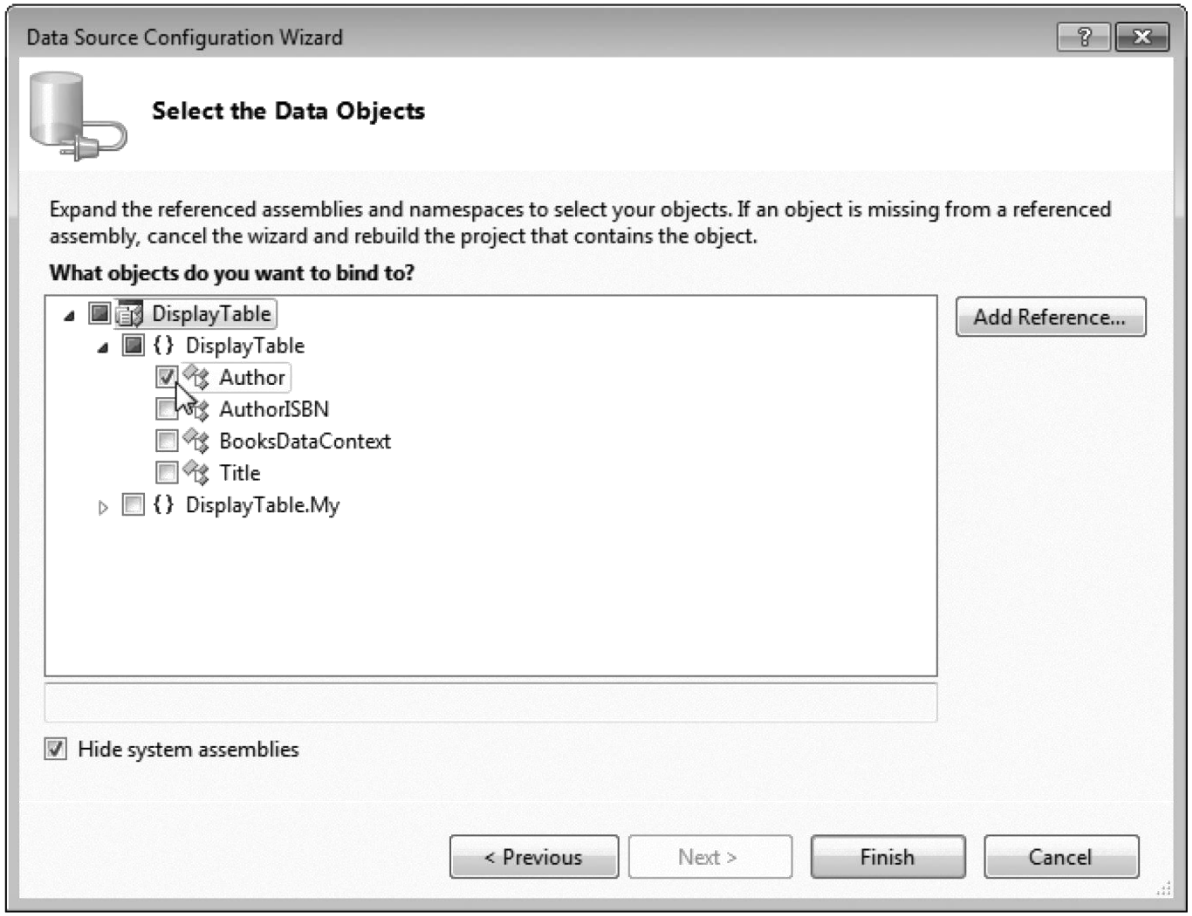


Fig. 18.31 | Selecting the `Author` LINQ to SQL class as the data source.

Fig. 18.32 | **Data Sources** window showing the `Author` class as a data source.

18.8.2 Data Bindings Between Controls and the LINQ to SQL Classes (Cont.)

Step 2: Creating GUI Elements

- Switch to Design view for the `DisplayAuthorsTable` class.
- Click the Author node in the Data Sources window—it should change to a drop-down list.
- Open the drop-down by clicking the down arrow and ensure that the `DataGridView` option is selected—this is the GUI control that will be used to display and interact with the data.
- Drag the `Author` node from the Data Sources window onto the `Form` in Design view.

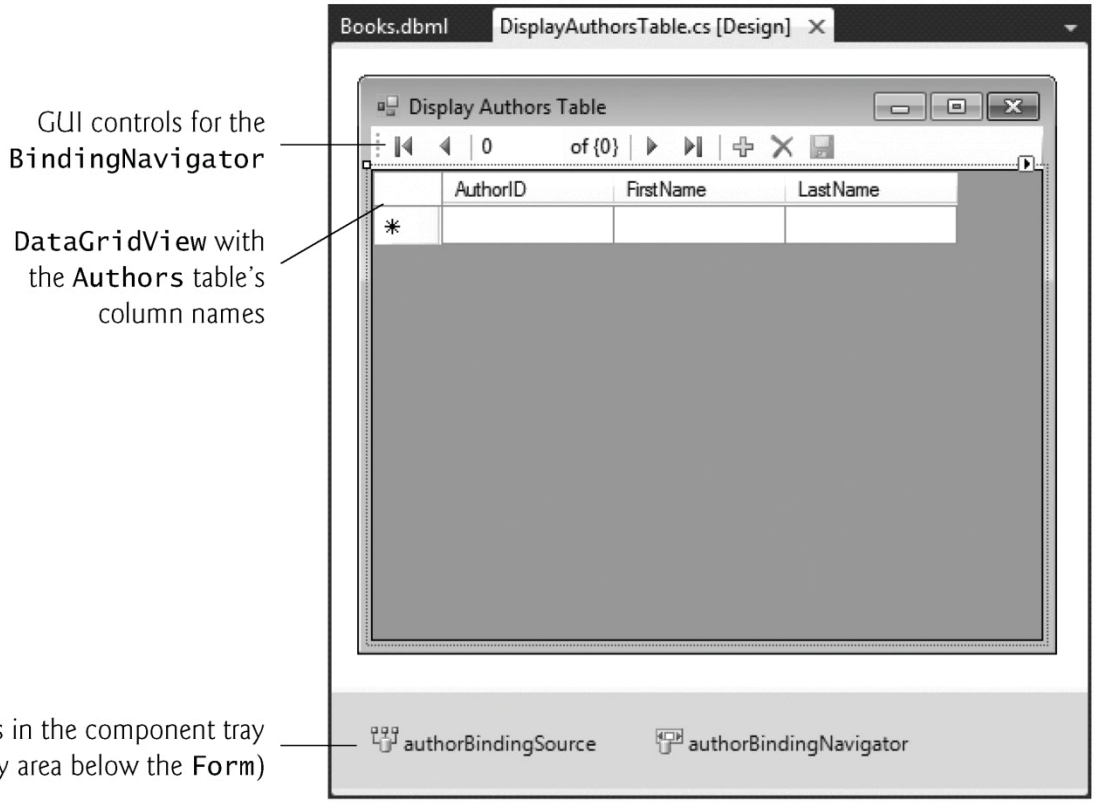


Fig. 18.33 | Component tray holds nonvisual components in Design view.

18.8.2 Data Bindings Between Controls and the LINQ to SQL Classes (Cont.)

Step 3: Connecting the `BooksDataContext` to the `authorBindingSource`

- The final step is to connect the `BooksDataContext` to the `authorBindingSource` (created earlier in this section), so that the application can interact with the database.

```
1  // Fig. 18.34: DisplayAuthorsTable.cs
2  // Displaying data from a database table in a DataGridView.
3  using System;
4  using System.Linq;
5  using System.Windows.Forms;
6
7  namespace DisplayTable
8  {
9     public partial class DisplayAuthorsTable : Form
10    {
11       // constructor
12       public DisplayAuthorsTable()
13       {
14          InitializeComponent();
15       } // end constructor
16
17       // LINQ to SQL data context
18       private BooksDataContext database = new BooksDataContext();
19
```

Fig. 18.34 | Displaying data from a database table in a DataGridView. (Part 1 of 3.)

```
20          // load data from database into DataGridView
21          private void DisplayAuthorsTable_Load( object sender, EventArgs e )
22          {
23             // use LINQ to order the data for display
24             authorBindingSource.DataSource =
25                from author in database.Authors
26                orderby author.AuthorID
27                select author;
28          } // end method DisplayAuthorsTable_Load
29
30          // click event handler for the Save Button in the
31          // BindingNavigator saves the changes made to the data
32          private void authorBindingNavigatorSaveItem_Click(
33             object sender, EventArgs e )
34          {
35             Validate(); // validate input fields
36             authorBindingSource.EndEdit(); // indicate edits are complete
37             database.SubmitChanges(); // write changes to database file
38          } // end method authorBindingNavigatorSaveItem_Click
39       } // end class DisplayAuthorsTable
40    } // end namespace DisplayTable
```

Fig. 18.34 | Displaying data from a database table in a `DataGridView`. (Part 2 of 3.)

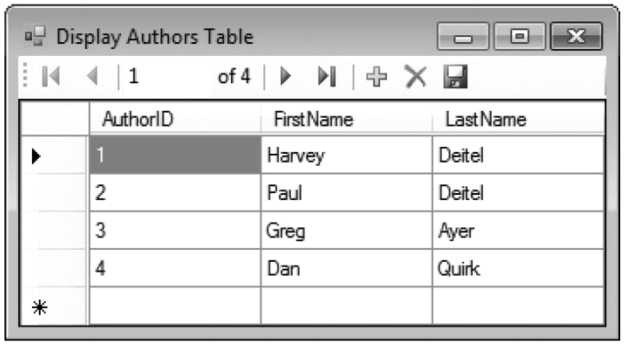


Fig. 18.34 | Displaying data from a database table in a `DataGridView`. (Part 3 of 3.)

18.8.2 Data Bindings Between Controls and the LINQ to SQL Classes (Cont.)

Step 4: Saving Modifications Back to the Database

- By default, the `BindingNavigator`'s Save Data `Button` is disabled. To enable it, right click the icon and select Enabled.
- Double click the icon to create its `Click` event handler.

18.8.2 Data Bindings Between Controls and the LINQ to SQL Classes (Cont.)

Step 5: Configuring the Database File to Persist Changes

- To persist changes for all executions, select the database in the Solution Explorer and set the Copy to Output Directory property in the Properties window to Copy if newer.

18.9 Dynamically Binding LINQ to SQL Query Results

The Display Query Results application allows the user to select a query from the `ComboBox` at the bottom of the window, then displays the results of the query.

a) Results of the "All titles" query, which shows the contents of the `Titles` table ordered by the book titles

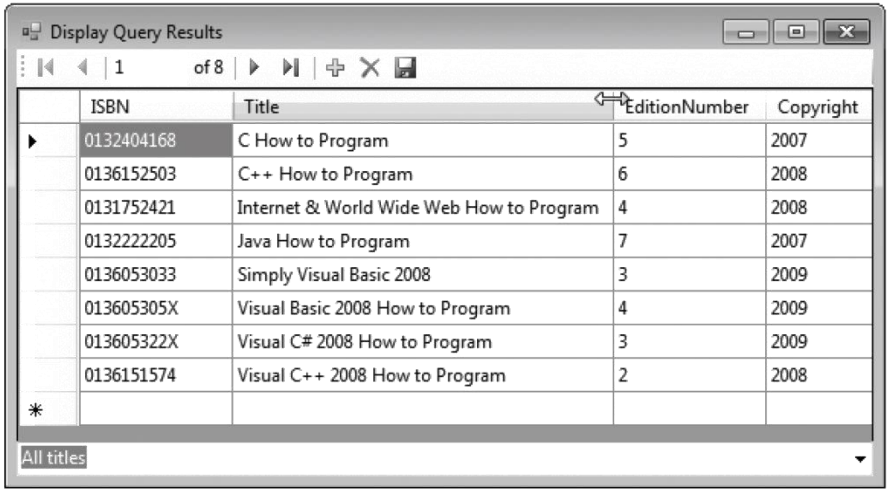
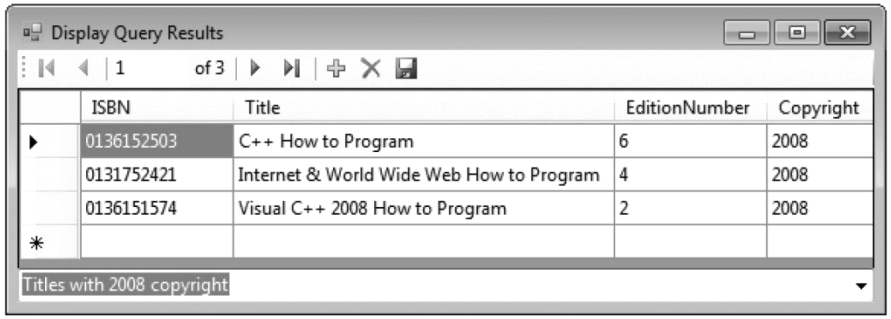


Fig. 18.35 | Sample execution of the Display Query Results application.

b) Results of the "Titles with 2008 copyright" query



c) Results of the "Titles ending with 'How to Program'" query

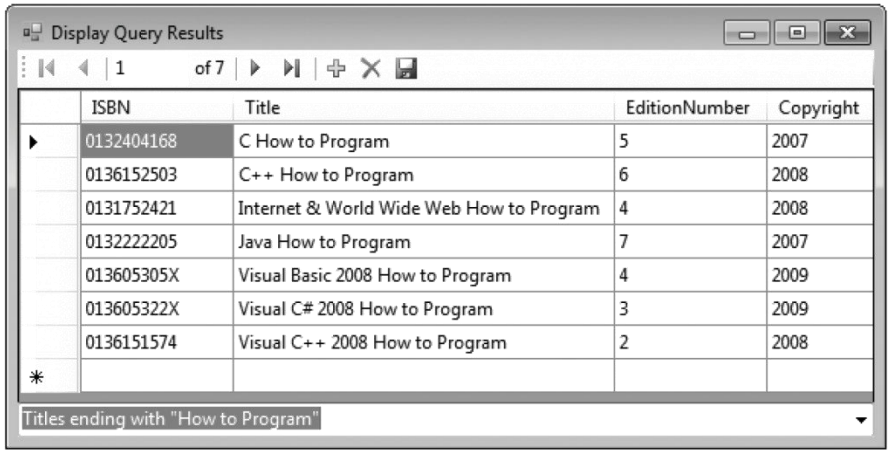


Fig. 18.35 | Sample execution of the **Display Query Results** application.

18.9.1 Creating the Display Query Results GUI

Step 1: Creating the Project

- First, create a new Windows Forms Application named `DisplayQueryResult`. Rename the source file to `TitleQueries.cs`. Set the `Form`'s `Text` property to Display Query Results.

Step 2: Creating the LINQ to SQL Classes

- Follow the steps in to add the Books database to the project and generate the LINQ to SQL classes.

18.9.1 Creating the Display Query Results GUI (Cont.)

Step 3: Creating a DataGridView to Display the Titles Table

- Follow *Steps 1* and *2* in Section 18.8.2 to create the data source and the `DataGridView`.
- Select the `Title` class as the data source, and drag the `Title` node from the Data Sources window onto the form.

18.9.1 Creating the Display Query Results GUI (Cont.)

- ***Step 4: Adding a ComboBox to the Form***
- In Design view, add a `ComboBox` named `queriesComboBox` below the `DataGridView` on the `Form`.
- Set the `ComboBox`'s `Dock` property to `Bottom` and the `Data-GridView's Dock` property to `Fill`.
- Add the names of the queries to the `ComboBox`.
- Open the `ComboBox`'s String Collection Editor by right clicking the `ComboBox` and selecting Edit Items.
- You can also access the String Collection Editor from the smart tag menu.

18.9.1 Creating the Display Query Results GUI (Cont.)

- In the String Collection Editor, add the following three items to `queriesComboBox`—one for each of the queries we'll create:
  - All titles
  - Titles with 2008 copyright
  - Titles ending with "How to Program"

18.9.2 Coding the Display Query Results Application

- Next you must write code that executes the appropriate query each time the user chooses a different item from `queriesComboBox`.

```
1  // Fig. 18.36: TitleQueries.cs
2  // Displaying the result of a user-selected query in a DataGridView.
3  using System;
4  using System.Linq;
5  using System.Windows.Forms;
6
7  namespace DisplayQueryResult
8  {
9     public partial class TitleQueries : Form
10    {
11       public TitleQueries()
12       {
13          InitializeComponent();
14       } // end constructor
15
16       // LINQ to SQL data context
17       private BooksDataContext database = new BooksDataContext();
18
```

Fig. 18.36 | Displaying the result of a user-selected query in a DataGridView. (Part 1 of 4.)

```
19          // load data from database into DataGridView
20          private void TitleQueries_Load( object sender, EventArgs e )
21          {
22             // write SQL to standard output stream
23             database.Log = Console.Out;
24
25             // set the ComboBox to show the default query that
26             // selects all books from the Titles table
27             queriesComboBox.SelectedIndex = 0;
28          } // end method TitleQueries_Load
29
30          // Click event handler for the Save Button in the
31          // BindingNavigator saves the changes made to the data
32          private void titleBindingNavigatorSaveItem_Click(
33             object sender, EventArgs e )
34          {
35             Validate(); // validate input fields
36             titleBindingSource.EndEdit(); // indicate edits are complete
37             database.SubmitChanges(); // write changes to database file
38
39             // when saving, return to "all titles" query
40             queriesComboBox.SelectedIndex = 0;
41          } // end method titleBindingNavigatorSaveItem_Click
```

Fig. 18.36 | Displaying the result of a user-selected query in a DataGridView. (Part 2 of 4.)

```
42
43          // loads data into titleBindingSource based on user-selected query
44          private void queriesComboBox_SelectedIndexChanged(
45             object sender, EventArgs e )
46          {
47             // set the data displayed according to what is selected
48             switch ( queriesComboBox.SelectedIndex )
49             {
50                case 0: // all titles
51                   // use LINQ to order the books by title
52                   titleBindingSource.DataSource =
53                      from book in database.Titles
54                      orderby book.Title1
55                      select book;
56                   break;
57                case 1: // titles with 2008 copyright
58                   // use LINQ to get titles with 2008
59                   // copyright and sort them by title
60                   titleBindingSource.DataSource =
61                      from book in database.Titles
62                      where book.Copyright == "2008"
63                      orderby book.Title1
64                      select book;
65                   break;
```

Fig. 18.36 | Displaying the result of a user-selected query in a DataGridView. (Part 3 of 4.)

```
66              case 2: // titles ending with "How to Program"
67                 // use LINQ to get titles ending with
68                 // "How to Program" and sort them by title
69                 titleBindingSource.DataSource =
70                    from book in database.Titles
71                    where book.Title1.EndsWith( "How to Program" )
72                    orderby book.Title1
73                    select book;
74                 break;
75           } // end switch
76
77           titleBindingSource.MoveFirst(); // move to first entry
78        } // end method queriesComboBox_SelectedIndexChanged
79     } // end class TitleQueries
80  } // end namespace DisplayQueryResult
```

Fig. 18.36 | Displaying the result of a user-selected query in a `DataGridView`. (Part 4 of 4.)