6
Presenting Data with the DataGridView Control

THE PRECEDING CHAPTERS showed many detailed examples of data binding to simple bound controls and list bound controls. However, one of the most common ways of presenting data is in tabular form. Users are able to quickly scan and understand large amounts of data visually when it is presented in a table. In addition, users can interact with that data in a number of ways, including scrolling through the data, sorting the data based on columns, editing the data directly in the grid, and selecting columns, rows, or cells. In .NET 1.0, the `DataGrid` control was the primary Windows Forms control for presenting tabular data. Even though that control had a lot of capability and could present basic tabular data well, it was fairly difficult to customize many aspects of the control. Additionally, the `DataGrid` control didn't expose enough information to the programmer about the user interactions with the grid and changes occurring in the grid due to programmatic modifications of the data or formatting. Due to these factors and a large number of new features that customers requested, the Windows Client team at Microsoft decided to introduce a replacement control for the `DataGrid` in .NET 2.0. That new control, the `DataGridView` control, is the focus of this chapter.

DataGridView Overview

The `DataGridView` control is a very powerful, flexible, and yet easy-to-use control for presenting tabular data. It is far more capable than the `DataGrid` control and is easier to customize and interact with. You can let the grid do all the work of presenting data in tabular form by setting the data-binding properties on the control appropriately. You can also take explicit control of presenting data in the grid through the new features of unbound columns and virtual mode. **Unbound columns** let you formulate the contents of the cell as the cells are being added to the grid. **Virtual mode** gives you a higher degree of control by allowing you to wait until a cell is actually being displayed to provide the value it will contain.

You can make the grid act like a spreadsheet, so that the focus for interaction and presentation is at the cell level instead of at the row or column level. You can control the formatting and layout of the grid with fine-grained precision simply by setting a few properties on the control. Finally, you can plug in a number of predefined column and cell control types, or provide your own custom controls, and you can even mix different control types within different cells in the same row or column.

Figure 6.1 shows an example of a `DataGridView` control in action with some of the key visual features highlighted. You can see that the grid picks up the visual styles of Windows XP; they are much like many of the Windows Forms controls in .NET 2.0. The grid is composed of columns and rows, and the intersection of a column and a row is a cell. The cell is the basic unit of presentation within the grid, and is highly customizable in

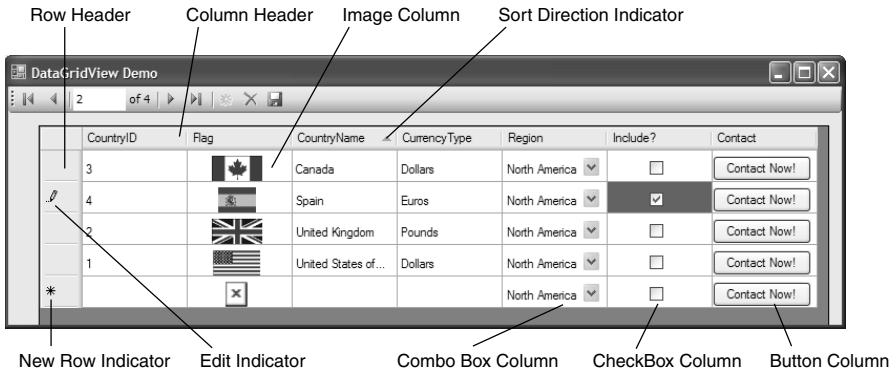

FIGURE 6.1: DataGridView in Action

appearance and behavior through the properties and events exposed by the grid. There are header cells for the rows and columns that can be used to maintain the context of the data presented in the grid. These header cells can contain graphical glyphs to indicate different modes or functions of the grid, such as sorting, editing, new rows, and selection. The grid can contain cells of many different types, and can even mix different cell types in the same column if the grid isn't data bound.

Basic Data Binding with the DataGridView

The easiest way to get started using the `DataGridView` control is to use it in basic data-binding scenarios. To do this, you first need to obtain a collection of data, typically through your business layer or data access layer. You then set the grid's data-binding properties to bind to the data collection, as described in Chapters 4 and 5. Just like with other Windows Forms controls, the recommended practice in .NET 2.0 is to always bind your actual client-side data source to an instance of the `BindingSource` class and then bind your controls to the binding source. The following code shows this process.

```
private void OnFormLoad(object sender, EventArgs e)
{
   // Create adapter to get data source
   CustomersTableAdapter adapter = new CustomersTableAdapter();
   // Bind table data source to binding source
   m_CustomersBindingSource.DataSource = adapter.GetData();
   // Bind the binding source to grid control
   m_Grid.DataSource = m_CustomersBindingSource;
}
```

Alternatively, if the binding source is bound to a collection of data collections, such as a data set, then you can refine what part of the data source you want to bind to using the `DataMember` property:

```
private void OnFormLoad(object sender, EventArgs e)
{
   // Create adapter to get data source
   CustomersTableAdapter adapter = new CustomersTableAdapter();
   // Get data set instance
   CustomersDataSet customers = new CustomersDataSet();
   // Fill data set
```

```
    adapter.Fill(customers);
    // Bind binding source to the data set
    m_CustomersBinding source.DataSource = customers;
    // Bind grid to the Customers table within the data source
    m_Grid.DataSource = m_CustomersBinding source;
    m_Grid.DataMember = "Customers";
}
```

For basic data-binding scenarios, the `DataGridView` functions exactly like the `DataGrid` control did in .NET 1.0, except that the combination of `DataSource` and `DataMember` must resolve to a collection of data items, such as a `DataTable` or object collection. Specifically, they need to resolve to an object that implements the `IList` interface.

The `DataGrid` could be bound to a collection of collections, such as a `DataSet`, and if so, the `DataGrid` presented hierarchical navigation controls to move through the collections of data. However, this capability was rarely used, partly because the navigation controls that were presented inside the `DataGrid` were a little unintuitive and could leave the user disoriented. As a result, the Windows Client team that developed the `DataGridView` control decided *not* to support hierarchical navigation within the control. The `DataGridView` is designed to present a single collection of data at a time. You can still achieve an intuitive hierarchical navigation through data, but you will usually use more than one control to do so, adopting a master-details approach as discussed in previous chapters.

The `DataSource` property can be set to any collection of objects that implements one of four interfaces: `IList`, `IListSource`, `IBindingList`, or `IBindingListView`. (These interfaces will be discussed in more detail in Chapter 7.) If the data source is itself a collection of data collections, such as a data set or an implementer of `IListSource`, then the `DataMember` property must identify which data collection within that source to bind to. If the `DataSource` property is set to an implementer of `IList` (from which both `IBindingList` and `IBindingListView` derive), then the `DataMember` property can be `null` (the default value). When you bind the `DataGridView` to a binding source, the `BindingSource` class itself implements `IBindingListView` (as well as several other data-binding related interfaces), so you can actually bind a grid to any kind of collection that a binding source can work with through a binding source, which includes simple collections that only implement `IEnumerable`.

Any time the `DataSource` and/or `DataMember` properties are set, the grid will iterate through the items found in the data collection and will refresh the data-bound columns of the grid. If the grid is bound to a binding source, any change to the underlying data source to which the binding source is bound also results in the data-bound columns in the grid being refreshed. This happens because of events raised from the binding source to any bound controls whenever its underlying collection changes.

Like most properties on the `DataGridView` control, any time the `DataSource` and `DataMember` properties are set, they fire the `DataSourceChanged` and `DataMemberChanged` events, respectively. This lets you hook up code that responds to the data binding that has changed on the grid. You can also react to the `DataBindingComplete` event, since that will fire after the data source or data member has changed and data binding has been updated. However, if you are trying to monitor changes in the data source, you usually are better off monitoring the corresponding events on the `BindingSource` component rather than subscribing to the events on the grid itself. This is especially true if the code you are using to handle the event affects other controls on the form. Because you should always bind your controls to a binding source instead of the data source itself if possible, the binding source is the best place to monitor changes in the data source.

Controlling Modifications to Data in the Grid

The `DataGridView` control gives you explicit control over whether users can edit, delete, or add rows in the grid. After the grid has been populated with data, the user can interact with the data presented in the grid in a number of ways, as discussed earlier. By default, those interactions include editing the contents of cells (fields) in a row, selecting a row and deleting it with the Delete keyboard key, or adding a new row using an empty row that displays as the last row in the grid.

If you want to disallow any of these interactions, set the `AllowUserToAddRows` or `AllowUserToDeleteRows` properties to `false`, or set the `ReadOnly` property to `true` for adding, deleting, or editing, respectively. Each of these properties also raise corresponding *`XXX`*`Changed` property changed events whenever their values are set. When you support adding, editing, or deleting, you may need to handle certain additional events to

accept new values for unbound rows or for virtual mode, as described later in this chapter.

Programmatic DataGridView Construction

The most common way of using the grid is with data-bound columns. When you bind to data, the grid creates columns based on the schema or properties of the data items, and generates rows in the grid for each data item found in the bound collection. If the data binding was set up statically using the designer (as has been done in most of the examples in this book), the types and properties of the columns in the grid were set at design time. If the data binding is being done completely dynamically, the `AutoGenerateColumns` property is `true` by default, so the column types are determined on the fly based on the type of the bound data items. You may want to create and populate a grid programmatically when working with a grid that contains only unbound data. To know what code you need to write, you need to know the `DataGridView` object model a little better.

The first thing to realize is that like all .NET controls, a grid on a form is just an instance of a class. That class contains properties and methods that you can use to code against its contained object model. For `DataGridView` controls, the object model includes two collections—`Columns` and `Rows`—which contain the objects that compose the grid. These objects are cells, or more specifically, objects derived from instances of `DataGridViewCell`. The `Columns` collection contains instances of `DataGridViewColumn` objects, and the `Rows` collection contains instances of `DataGridViewRows`.

Programmatically Adding Columns to a Grid

There are a number of ways to approach programmatically adding columns and rows to the grid. The first step is to define the columns from which the grid is composed. To define a column, you have to specify a cell template on which the column is based. The cell template will be used by default for the cells in that column whenever a row is added to the grid. Cell templates are instances of a `DataGridViewCell` derived class. You can use the .NET built-in cell types to present columns as text boxes, buttons, check boxes, combo boxes, hyperlinks, and images. Another built-in cell type renders the column headers in the grid. For each of the cell types,

there is a corresponding column type that is designed to contain that cell type. You can construct `DataGridViewColumn` instances that provide a cell type as a template, but in general you'll want to create an instance of a derived column type that is designed to contain the specific type of cell you want to work with. Additionally, you can define your own custom cell and column types (discussed later in this chapter).

For now, let's stick with the most common and simple cell type, a `DataGridViewTextBoxCell`—a text box cell. This also happens to be the default cell type. You can programmatically add a text box column in one of three ways:

- Use an overloaded version of the `Add` method on the `Columns` collection of the grid:

```
// Just specify the column name and header text
m_Grid.Columns.Add("MyColumnName", "MyColumnHeaderText");
```

- Obtain an initialized instance of the `DataGridViewTextBoxColumn` class. You can achieve this by constructing an instance of the `DataGridViewTextBoxCell` class and passing it to the constructor for the `DataGridViewColumn`, or just construct an instance of a `DataGridViewTextBoxColumn` using the default constructor. Once the column is constructed, add it to the `Columns` collection of the grid:

```
// Do this:
DataGridViewTextBoxColumn newCol = new DataGridViewTextBoxColumn();
// Or this:
DataGridViewTextBoxCell newCell = new DataGridViewTextBoxCell();
DataGridViewColumn newCol2 = new DataGridViewColumn(newCell);
// Then add to the columns collection:
m_Grid.Columns.Add(newCol);
m_Grid.Columns.Add(newCol2);
```

  If you add columns this way, their name and header values are null by default. To set these or other properties on the columns, you can access the properties on the column instance before or after adding it to the `Columns` collection. You could also index into the `Columns` collection to obtain a reference to a column, and then use that reference to get at any properties you need on the column.

- Set the grid's `ColumnCount` property to some value greater than zero. This approach is mainly used to quickly create a grid that only contains text box cells or to add more text box columns to an existing grid.

```
// Constructs five TextBox columns and adds them to the grid
m_Grid.ColumnCount = 5;
```

When you set the `ColumnCount` property like this, the behavior depends on whether there are already any columns in the grid. If there are existing columns and you specify fewer than the current number of columns, the `ColumnCount` property will remove as many columns from the grid as necessary to create only the number of columns you specify, starting from the rightmost column and moving to the left. This is a little destructive and scary because it lets you delete columns without explicitly saying which columns to eliminate, so I recommend to avoid using the `ColumnCount` property to remove columns.

However, when adding text box columns, if you specify more columns than the current number of columns, additional text box columns will be added to the right of the existing columns to bring the column count up to the number you specify. This is a compact way to add some columns for dynamic situations.

Programmatically Adding Rows to a Grid

Once you have added columns to the grid, you can programmatically add rows to the grid as well. In most cases, this involves using the `Add` method of the `Rows` collection on the grid. When you add a row this way, the row is created with each cell type based on the cell template that was specified for the column when the columns were created. Each cell will have a default value assigned based on the cell type, generally corresponding to an empty cell.

```
// Add a row
m_Grid.Rows.Add();
```

Several overloads of the `Add` method let you add multiple rows in a single call or pass in a row that has already been created. The `DataGridView` control also supports creating heterogeneous columns, meaning that the

column can have cells of different types. To create heterogeneous columns, you have to construct the row first without attaching it to the grid. You then add the cells that you want to the row, and then add the row to the grid. For example, the following code adds a combo box as the first cell in the row, adds some items to the combo box, adds text boxes for the remaining four cells, and then adds the row to the grid.

```
private void OnAddHeterows(object sender, EventArgs e)
{
   m_Grid.ColumnCount = 5; // Create 5 text box columns
   DataGridViewRow heterow = new DataGridViewRow();
   DataGridViewComboBoxCell comboCell = new
DataGridViewComboBoxCell();
   comboCell.Items.Add("Black");
   comboCell.Items.Add("White");
   comboCell.Value = "White";
   heterow.Cells.Add(comboCell);
   for (int i = 0; i < 4; i++)
   {
      heterow.Cells.Add(new DataGridViewTextBoxCell());
   }
   m_Grid.Rows.Add(heterow); // this row has a combo in first cell
}
```

To add a row to a grid this way, the grid must already have been initialized with the default set of columns that it will hold. Additionally, the number of cells in the row that is added must match that column count. In this code sample, five text box columns are implicitly added by specifying a column count of five, then the first row is added as a heterogeneous row by constructing the cells and adding them to the row before adding the row to the grid.

You can also save yourself some code by using the grid's existing column definitions to create the default set of cells in a row using the `CreateCells` method, then replace just the cells that you want to be different from the default:

```
DataGridViewRow heterow = new DataGridViewRow();
heterow.CreateCells(m_Grid);
heterow.Cells.RemoveAt(0);
heterow.Cells.Insert(0, new DataGridViewComboBoxCell());
m_Grid.Rows.Add(heterow);
```

To access the contents of the cells programmatically, you index into the `Rows` collection on the grid to obtain a reference to the row, and then index into the `Cells` collection on the row to access the cell object.

Once you have a reference to the cell, you can do anything with that cell that the actual cell type supports. If you want to access specific properties or methods exposed by the cell type, you have to cast the reference to the actual cell type. To change a cell's contents, you set the `Value` property to an appropriate value based on the type of the cell. What constitutes an appropriate value depends on what kind of cell it is. For text box, link, button, and header cell types, the process is very similar to what was described for the `Binding` object in Chapter 4. Basically, the value you set on the `Value` property of the cell needs to be convertible to a string, and formatting will be applied in the painting process. To change the formatting of the output string, set the `Format` property of the style used for the cell. The style is an instance of a `DataGridViewCellStyle` object and is exposed as another property on the cell, not surprisingly named `Style`. The cell's style contains other interesting properties that you can set (described later in the section "Formatting with Styles").

For example, if you want to set the contents of a text box cell to the current date using the short date format, you could use the following code:

```
m_Grid.Rows[0].Cells[2].Value = DateTime.Now;
m_Grid.Rows[0].Cells[2].Style.Format = "d";
```

This sets the value of the third cell in the first row to an instance of a `DateTime` object, which is convertible to a string, and sets the format string to the short date predefined format string "d" (which is the short date format—MM/YYYY). When that cell gets rendered, it will convert the stored `DateTime` value to a string using the format string.

Custom Column Content with Unbound Columns

Now that you understand how to programmatically create columns and rows, and populate them with values, you may be wondering if you have to go to all that trouble any time you want to present content in a cell that isn't bound to data. The good news is that there is a faster way for most

scenarios where you want to present unbound data. You will need to programmatically create all the columns in the grid (although you can get the designer to write this code for you too, as shown later), but you can use events to make populating the values a little easier, especially when you mix bound data with unbound columns.

An **unbound column** is a column that isn't bound to data. You add unbound columns to a grid programmatically, and you populate the column's cells either programmatically as shown in the previous section or by using events as shown in this section. You can still add columns to the grid that are automatically bound to columns or properties in the data items of the data source. You do this by setting the `DataPropertyName` property on the column after it is created. Then you can add unbound columns as well. The rows of the grid will be created when you set the grid's `DataSource` property to the source as in the straight data-binding case, because the grid will iterate through the rows or objects in the data collection, creating a new row for each.

There are two primary ways to populate the contents of unbound columns: handling the `RowsAdded` event or handling the `CellFormatting` event. The former is a good place to set the cell's value to make it available for programmatic retrieval later. The latter is a good place to provide a value that will be used for display purposes only and won't be stored as part of the data retained by the grid cells collection. The `CellFormatting` event can also be used to transform values as they are presented in a cell to something different than the value that is actually stored in the data that sits behind the grid.

To demonstrate this capability, let's look at a simple example.

1. Start a new Windows application project in Visual Studio 2005, and add a new data source to the project for the Customers table in the Northwind database (this is described in Chapter 1—here are the basic steps):
   a. Select Data > Add New Data Source.
   b. Select *Database* as the data source type.
   c. Select or create a connection to the Northwind database.
   d. Select the *Customers* table from the tree of database objects.
   e. Name the data set **CustomersDataSet** and finish the wizard.

At this point you have an empty Windows Forms project with a typed data set for Customers defined.

2. Add a `DataGridView` and `BindingSource` to the form from the Toolbox, naming them `m_CustomersGrid` and `m_Customers-BindingSource` respectively.
3. Add the code in Listing 6.1 to the constructor for the form, following the call to `InitializeComponents`.

LISTING 6.1: Dynamic Column Construction

```
public Form1()
{
   InitializeComponent();

   // Get the data
   CustomersTableAdapter adapter = new CustomersTableAdapter();
   m_CustomersBindingSource.DataSource = adapter.GetData();

   // Set up the grid columns
   m_CustomersGrid.AutoGenerateColumns = false;
   int newColIndex = m_CustomersGrid.Columns.Add("CompanyName",
      "Company Name");
   m_CustomersGrid.Columns[newColIndex].DataPropertyName =
      "CompanyName";
   newColIndex = m_CustomersGrid.Columns.Add("ContactName",
      "Contact Name");
   m_CustomersGrid.Columns[newColIndex].DataPropertyName =
      "ContactName";
   newColIndex = m_CustomersGrid.Columns.Add("Phone","Phone");
   m_CustomersGrid.Columns[newColIndex].DataPropertyName = "Phone";
   newColIndex = m_CustomersGrid.Columns.Add("Contact", "Contact");

   // Subscribe events
   m_CustomersGrid.CellFormatting += OnCellFormatting;
   m_CustomersGrid.RowsAdded += OnRowsAdded;

   // Data bind
   m_CustomersGrid.DataSource = m_CustomersBindingSource;
}
```

This code first retrieves the Customers table data using the table adapter's `GetData` method. As discussed earlier in the book, the table adapter was created along with the typed data set when you added the data source to your project. It sets the returned data table

as the data source for the binding source. The `AutoGenerateColumns` property is set to `false`, since the code programmatically populates the columns collection. Then four text box columns are added to the grid using the overload of the `Add` method on the `Columns` collection, which takes a column name and the header text. The first three columns are set up for data binding to the Customers table's CompanyName, ContactName, and Phone columns by setting the `DataPropertyName` property on the column after it is created. The fourth column is the unbound column and is simply created at this point through the call to the `Add` method. It will be populated later through events.

Finally, the events of interest are wired up to the methods that will handle them using delegate inference. Using this new C# feature, you don't have to explicitly create an instance of a delegate to subscribe a handler for an event. You just assign a method name that has the appropriate signature for the event's delegate type, and the compiler will generate the delegate instance for you. In Visual Basic, you use the `AddHandler` operator, which has always operated similarly.

When the data source is set on the grid and the grid is rendered, the grid iterates through the rows of the data source and adds a row to the grid for each data source row, setting the values of the bound column cells to the corresponding values in the data source. As each row is created, the `RowsAdded` event is fired. In addition, a series of events are fired for every cell in the row as it is created.

4. Add the following method as the handler for the `CellFormatting` event:

```
private void OnCellFormatting(object sender,
   DataGridViewCellFormattingEventArgs e)
{
   if (e.ColumnIndex == m_CustomersGrid.Columns["Contact"].Index)
   {
      e.FormattingApplied = true;
      DataGridViewRow row = m_CustomersGrid.Rows[e.RowIndex];
      e.Value = string.Format("{0} : {1}",
         row.Cells["ContactName"].Value,
         row.Cells["Phone"].Value);
   }
}
```

As previously mentioned, you can use the `CellFormatting` event if you are programmatically setting the displayed values for the cells. The event argument that is passed in to the `CellFormatting` event exposes a number of properties to let you know what cell is being rendered. You can use the `ColumnIndex` to determine which column the event is being fired for. It is checked against the index of the Contact column using a lookup in the `Columns` collection.

Once it is determined that this is the column you want to supply a programmatic value for, you can obtain the actual row being populated using the `RowIndex` property on the event argument. In this case, the code just concatenates the ContactName and Phone from the row to form a contact information string using the `String.Format` method, and sets that string as the value on the Contact column.

In other situations, you may use the `CellFormatting` event to do something like look up a value from another table, such as using a foreign key, and use the retrieved value as the displayed value in the unbound column. It also sets the `FormattingApplied` property on the event argument to `true`. This is very important to do; it signals the grid that this column is being dynamically updated. If you fail to do this, you will get an infinite loop if you also set the column to have its size automatically determined, as discussed in a later section.

> **NOTE** **Always Set `FormattingApplied` to True When Dynamically Formatting Cell Content**
>
> If you fail to set the `FormattingApplied` property on the event argument to the `CellFormatting` event, the grid won't know that the content is being determined dynamically and may not work properly for certain other grid behaviors.

It should be noted that the code example for the `CellFormatting` event is fairly inefficient from a performance perspective. First, you wouldn't want to look up the column's index by name in the column collection every time. It would be more efficient to look it up once, store it in a

member variable, and then use that index directly for comparison. I went with the lookup approach in this sample to keep it simple and easy to read—so you could focus on the grid details instead of a bunch of performance-oriented code. Besides, this is a somewhat contrived example anyway; it's just meant to demonstrate how to create an unbound column.

If you want to set the actual stored cell values of unbound columns in the grid, a better way to do this is to handle the `RowsAdded` event. As you might guess from the name, this event is fired as rows are added to the grid. By handling this event, you can populate the values of all the unbound cells in a row in one shot, which is slightly more efficient than having to handle the `CellFormatting` event for every cell. The `RowsAdded` event can be fired for one row at a time if you are programmatically adding rows in a loop, or it can be fired just once for a whole batch of rows being added, such as when you data bind or use the `AddCopies` method of the rows collection. The event argument to `RowsAdded` contains properties for `RowIndex` and `RowCount`; these properties let you iterate through the rows that were added to update the unbound column values in a loop. The following method shows an alternative approach for populating the grid's Contact column from the previous example, using the `RowsAdded` method instead of the `CellFormatting` event:

```
private void OnRowsAdded(object sender,
   DataGridViewRowsAddedEventArgs e)
{
   for (int i = 0; i < e.RowCount; i++)
   {
      DataGridViewRow row = m_CustomersGrid.Rows[e.RowIndex + i];
      row.Cells["Contact"].Value = string.Format("{0} : {1}",
         row.Cells["ContactName"].Value,
         row.Cells["Phone"].Value);
   }
}
```

This code obtains the current row being set in the loop by indexing into the Rows collection with the `RowIndex` property from the event argument and a loop index offset up to the number of rows that are affected when this event fires. It then uses the existing data in that row's other columns to

compute the contents of the current row. For a real-world application, you could obtain or compute the value of the unbound columns in the row in the loop instead.

TIP Use the `CellFormatting` event to control displayed cell content and the `RowsAdded` event to control stored cell content

The `CellFormatting` event is fired for every cell as it is rendered, giving you a chance to inspect and possibly modify the value that will be presented for any of the cells in the grid. If you want to modify the actual value of an unbound column that is stored in the grid cells collection, use the `RowsAdded` event to set all the unbound column values in one shot when the row is added, instead of requiring a separate event firing for each cell. If the data that you are presenting is purely computed data, instead of data that you are retrieving or deriving from bound data, consider using virtual mode to populate those cells, as discussed in the next section.

NOTE Generate Computed Columns Where It Makes Sense

The example presented in this section simply computed a new column based on the contents of other columns. If this was needed just for display purposes, then using an unbound column might be the best approach. However, if that computed column value might be needed anywhere else in your application, it would make more sense to make it part of your application data entities by making it a custom data column based on an expression in the data set itself.

It is easy to add a custom column to a typed data set table using the data set designer, and there are a number of custom expressions to compute that column's values from other data in the table. If you need to generate the column data based on other tables or data in the database, doing so in a stored procedure might make even more sense.

The bottom line is that you need to pick the appropriate scope for your computed data. Don't use unbound columns with repeated code to display something that could have been computed once and made an actual member of the data you are working with.

Displaying Computed Data in Virtual Mode

Another scenario where you may want to take explicit control over providing a display value for each cell is when you are working with data that contains computed values, especially in combination with bound data and large sets of data. For example, you may need to present a collection of data that has tens of thousands or even millions of rows in a grid. You really ought to question the use case first before supporting such a thing, but if you really need to do this, you probably don't want to clobber your system's memory by creating multiple copies of that data. You may not even want to bring it all into memory at one time, especially if this data is being dynamically created or computed. However, you will want users to able be to smoothly scroll through all that data to locate the items of interest.

Virtual mode for the `DataGridView` lets you display values for cells as they are rendered, so that data doesn't have to be held in memory when it's not being used. With virtual mode you can specify which columns the grid contains at design time, and then provide the cell values as needed, so they display at runtime but not before. The grid will internally only maintain the cell values for the cells that are displayed; you provide the cell value on an as-needed basis.

When you choose to use virtual mode, you can either provide all of the row values through virtual mode event handling, or you can mix the columns of the grid with data-bound and unbound columns. You have to define the columns that will be populated through virtual mode as described earlier in the "Programmatic DataGridView Construction" section. If you also data bind some columns, then the rows will be populated through data binding and you just handle the events necessary for virtual mode for the columns that aren't data bound. If you aren't data binding, you have to add as many rows as you expect to present in the grid, so the grid knows to scale the scrollbar appropriately. You then need a way to get the values corresponding to virtual mode columns when they are needed for presentation. You can do this by computing the values dynamically when they are needed, which is one of the main times that you would use virtual mode. You might also use cached client-side data in the form of an

object collection or data set, or you might actually make round-trips to the server to get the data as needed. With the latter approach, you'll need a smart preload and caching strategy, because you could quickly bog down the application if data queries have to run between the client and the server while the user is scrolling. If the data being displayed is computed data, then it really makes sense to wait to compute values until they are actually going to be displayed.

Setting Up Virtual Mode

The following steps describe how to set up virtual mode data binding.

1. Create a grid and define the columns in it that will use virtual mode.
2. Put the grid into virtual mode by setting the `VirtualMode` property to `true`.
3. If you aren't using data binding, add as many rows to the grid as you want to support scrolling through. The easiest and fastest way to do this is to create one prototype row, then use the `AddCopies` method of the `Rows` collection to add as many copies of that prototype row as you would like. You don't have to worry about the cell contents at this point, because you are going to provide those dynamically through an event handler as the grid is rendered.
4. The final step is to wire up an event handler for the `CellValue-Needed` event on the grid. This event will only be fired when the grid is operating in virtual mode, and will be fired for each unbound column cell that is currently visible in the grid when it is first displayed and as the user scrolls.

The code in Listing 6.2 shows a simple Windows Forms application that demonstrates the use of virtual mode. A `DataGridView` object was added to the form through the designer and named `m_Grid`, and a button added to the form for checking how many rows had been visited when scrolling named `m_GetVisitedCountButton`.

LISTING 6.2: Virtual Mode Sample

```
partial class VirtualModeForm : Form
{
   private List<DataObject> m_Data = new List<DataObject>();
   private List<bool> m_Visited = new List<bool>();
   public VirtualModeForm()
   {
      InitializeComponent();
      m_Grid.CellValueNeeded += OnCellValueNeeded;
      m_GetVisitedCountButton.Click += OnGetVisitedCount;
      InitData();
      InitGrid();
   }

   private void InitData()
   {
      for (int i = 0; i < 1000001; i++)
      {
         m_Visited.Add(false);
         DataObject obj = new DataObject();
         obj.Id = i;
         obj.Val = 2 * i;
         m_Data.Add(obj);
      }
   }

   private void InitGrid()
   {
      m_Grid.VirtualMode = true;
      m_Grid.ReadOnly = true;
      m_Grid.AllowUserToAddRows = false;
      m_Grid.AllowUserToDeleteRows = false;
      m_Grid.ColumnCount = 3;
      m_Grid.Rows.Add();
      m_Grid.Rows.AddCopies(0, 1000000);
     // Uncomment the next line and comment out the
     // the rest of the method to switch to data bound mode
     //    m_Grid.DataSource = m_Data;
   }
   private void OnCellValueNeeded(object sender,
      DataGridViewCellValueEventArgs e)
   {
      m_Visited[e.RowIndex] = true;
      if (e.ColumnIndex == 0)
      {
         e.Value = m_Data[e.RowIndex].Id;
      }
      else if (e.ColumnIndex == 1)
      {
```

continues

```
            e.Value = m_Data[e.RowIndex].Val;
            }
            else if (e.ColumnIndex == 2)
            {
               Random rand = new Random();
               e.Value = rand.Next();
            }
         }

         private void OnGetVisitedCount(object sender, EventArgs e)
         {
            int count = 0;
            foreach (bool b in m_Visited)
            {
               if (b) count++;
            }
            MessageBox.Show(count.ToString());
         }
      }
      public class DataObject
      {
         private int m_Id;
         private int m_Val;

         public int Val
         {
            get { return m_Val; }
            set { m_Val = value; }
         }

         public int Id
         {
            get { return m_Id; }
            set { m_Id = value; }
         }
      }
```

The form constructor starts by calling `InitializeComponent` as usual, to invoke the code written by the designer from drag-and-drop operations and Properties window settings. For this sample, that just declares and creates the grid and the button controls on the form. The constructor code then subscribes two event handlers using delegate inference.

The first event handler is the important one for virtual mode—the `CellValueNeeded` event. As mentioned earlier, this event is only fired when the grid is in virtual mode and is called for each unbound column cell that is visible in the grid at any given time. As the user scrolls, this event fires again

for each cell that is revealed through the scrolling operation. The constructor also subscribes a handler for the button click, which lets you see how many rows the `CellValueNeeded` event handler was actually called for.

After that, the constructor calls the `InitData` helper method, which creates a collection of data using a `List<T>` generic collection that contains instances of a simple `DataObject` class, defined at the end of Listing 6.2. The `DataObject` class has two integer values, `Id` and `Val`, which are presented in the grid. The collection is populated by the `InitData` helper method with one million rows. The `Id` property of each object is set to its index in the collection, and the `Val` property is set to two times that index.

> **TIP Favor object collections for large data collections**
>
> Object collections that are easily defined using the generic collections in .NET 2.0 are inherently lighterweight and perform better than storing your data in a data set. Typed data sets are a great way to go for smaller sets of data, particularly when you need to support updates, inserts, and deletes. But for simply presenting a large collection of data, you will pay a significant performance and memory cost for using a data set compared to a custom object collection.

Choosing Data Sets or Custom Collections

When you are dealing with very large collections of data in memory like the one in Listing 6.2, one of your primary concerns should be memory impact. Based on the other samples presented in this book so far, you may be inclined to use a typed data set as your data collection object. However, you should think twice before doing this. Even though improvements in the `DataSet` class (and therefore derived typed data set classes) in .NET 2.0 significantly improve its scalability and performance for large data sets, the `DataSet` class is still a fairly heavyweight object because of all the hierarchical, change tracking, and updating capability that is built into a data set.

continues

If you are thinking of presenting millions of rows of data in a grid, and you expect to let users edit the rows in the grid and let the data set updating capabilities propagate those changes back to your data store, then the `DataSet` class may actually be what you need. However, I would discourage you from designing your application this way. Displaying large collections of data should be a special use case that focuses on presentation—you are just letting the user scroll through large numbers of rows to locate a data item of interest. If you need to support editing of those data items, I suggest you send the user to a different form for editing. However, you can let them edit the values in the grid without needing the extra overhead of a data set, as described shortly.

Object collections like the `List<T>` generic collection are significantly more lightweight than a data set and result in less memory consumption and quicker construction times. If you are going to cache a large collection in memory, you should try to design the form to use a lightweight object collection to cache the values for display only.

Initializing the Grid

After the data is initialized, the constructor calls the `InitGrid` helper method, which does the following:

- Sets the grid into virtual mode.
- Turns off editing, adding, and deleting.
- Adds three text box columns to the grid by setting the `ColumnCount` property.
- Adds one row to the grid as a template.
- Uses the `AddCopies` method on the `Rows` collection to add one million more rows. This method also contains a commented-out line of code that can be used to change the VirtualMode download sample to be data bound against the object collection so that you can see the difference in load time and memory footprint.

After that, Windows Forms event handling takes over for the rest of the application lifecycle. Because the grid was set to virtual mode, the next thing that happens is the `OnCellValueNeeded` handler will start to get called for each cell that is currently displayed in the grid. This method is coded to extract the appropriate value from the data collection based on the row index and column index of the cell that is being rendered for the first two columns. For the third column, it actually computes the value of the cell on the fly, using the `Random` class to generate random numbers. It also sets a flag in the `m_Visited` collection—you can use this to see how many rows are actually being rendered when users scroll around with the application running.

Understanding Virtual Mode Behavior

If you run the VirtualMode sample application from Listing 6.2, note that as you run the mouse over the third column in the grid, the random numbers in the cells that the mouse passes over change. This happens because the `CellValueNeeded` event handler is called every time the cell paints, not just when it first comes into the scrolling region, and the `Random` class uses the current time as a seed value for computing the next random number. So if the values that will be calculated when `CellValueNeeded` are time variant, you will probably want to develop a smarter strategy for computing those values and caching them to avoid exposing changing values in a grid just because the mouse passes over them.

The `OnGetVisitedCount` button `Click` handler displays a dialog that shows the number of rows rendered based on the `m_Visited` collection. If you run the VirtualMode sample application, you can see several things worth noting about virtual mode. The first is that the biggest impact to runtime is the loading and caching of the large data collection on the client side. As a result, this is the kind of operation you would probably want to consider doing on a separate thread in a real application to avoid tying up the UI while the data loads. Using a `BackgroundWorker` component would be a good choice for this kind of operation.

When dealing with very large data sets, if the user drags the scrollbar thumb control, a large numbers of rows are actually skipped through the paging mechanisms and latency of the scroll bar. As a result, you only have

to supply a tiny percentage of the actual cell values unless the user does an extensive amount of scrolling in the grid. This is why virtual mode is particularly nice for computed values: you can avoid computing cell values that won't be displayed.

If you run this example and scroll around for a bit, then click the *Get Visited Count* button, you will see how many rows were actually loaded. For example, I ran this application and scrolled through the data from top to bottom fairly slowly several times. While doing so, I saw smooth scrolling performance that looked like I was actually scrolling through the millions of rows represented by the grid. However, in reality, only about 1,000 rows were actually rendered while I was scrolling.

What if you want to support editing of the values directly in the grid? Maybe you are just using virtual mode to present a computed column with a relatively small set of data, and you want to use that column's edited value to perform some other computation or store the edited value. Another event, `CellValuePushed`, is fired after an edit is complete on a cell in a virtual mode grid. If the grid doesn't have `ReadOnly` set to true, and the cells are of a type that supports editing (like a text box column), then the user can click in a cell, hesitate, then click again to put the cell into editing mode. After the user has changed the value and the focus changes to another cell or control through mouse or keyboard action, the `CellValuePushed` event will be fired for that cell. In an event handler for that event, you can collect the new value from the cell and do whatever is appropriate with it, such as write it back into your data cache or data store.

Virtual Mode Summary

That's all there is to virtual mode: Set the `VirtualMode` property to `true`, create the columns and rows you want the grid to have, and then supply a handler for the `CellValueNeeded` event that sets the appropriate value for the cell being rendered. If you need to support the editing of values directly in the grid, then also handle the `CellValuePushed` event and do whatever is appropriate with the modified values as the user makes the changes. Hopefully, you won't need to use virtual mode often in your applications, but it's nice to have for presenting very large data collections or computing column values on the fly. There are no hard and fast rules on

when virtual mode will be needed. If you are having scrolling performance problems in your application, or you want to avoid the memory impact of holding computed values for large numbers of rows in memory, you can see if virtual mode solves your problems. You will still need to think about your data retrieval and caching strategy, though, to avoid seriously hampering the performance of your application on the client machine.

Using the Built-In Column Types

Using a text box column is straightforward enough: you data bind to something that can be rendered as text, or set the `Value` property on a cell to something that can be converted to a string, and you are done. Using some of the other cell types may not be as easy to figure out, so this section steps through each of the built-in column types, pointing out its capabilities and how to use it.

The first thing to realize is that even though most of the functionality is surfaced at the cell level in a `DataGridView` and it can support spreadsheet-like behavior (as described later in this chapter), the grid is still primarily a tabular control. The columns in the grid usually represent information that can be determined at design time—specifically, the schema of the data that will be presented. The rows are usually determined dynamically at runtime and map to the structure specified by the columns. You may occasionally programmatically create columns for rendering based on dynamic data schemas at runtime, but even then you are first defining the data's shape (the columns) and then providing the data (the rows).

As a result, for each of the built-in cell types that the grid is capable of displaying, there is a corresponding column type designed to contain cells of that type. Each cell type is derived from the `DataGridViewCell` class, and each of the corresponding column types is derived from `DataGridViewColumn`. Each of the column types expose properties to aid in the data's data binding, and each column type corresponds to the expected content for the type of cells that the column contains. Likewise, each derived cell type may expose additional properties based on the type of content it is designed to display.

Because each built-in column type is different in subtle ways, it's best to cover them one at a time. However, since all of the cell types contained by the column types derive from the same base class, there are a number of properties from the base class that you'll use for controlling and accessing cell content. The properties of the `DataGridViewCell` base class are described in Table 6.1.

TABLE 6.1: DataGridViewCell Properties

Property Name	Type	Description
`ColumnIndex`	`Int32`	Gets the position index of the containing column within the grid.
`ContentBounds`	`Rectangle`	Gets the bounding rectangle for the content of the cell. The upper left corner will be relative to the upper left corner of the cell, and the width and height represent the area available for rendering content within the cell.
`ContextMenuStrip`	`ContextMenu-Strip`	Gets or sets the context menu associated with the cell. If the user right-clicks in the cell, this property's context menu displays. If a context menu has been assigned at the column level, setting a different one at the cell level will replace the one used for the rest of the column, but only for this cell.
`DefaultNew-RowValue`	`Object`	Gets the value that will be used if no value has been provided for the cell. The base class returns `null` from this property, but derived classes can provide a more meaningful value based on their expected content. For example, the image cell type returns a red X bitmap when no image has been provided for a value.

TABLE 6.1: DataGridViewCell Properties (Continued)

Property Name	Type	Description
`Displayed`	`Boolean`	True if the cell is currently displayed in the viewable area of the grid, false otherwise. This is a read-only property.
`EditedFormatted-Value`	`Object`	Gets the formatted version of the transient edited value of the cell, after any formatting or type conversion has been applied, and before the edited value has been made the current value through a focus change to a different cell.
`EditType`	`Type`	Gets the type of the control that will be rendered in the cell when in editing mode.
`ErrorIconBounds`	`Rectangle`	Gets the rectangle that bounds where the error icon will be presented so you can do any custom rendering based on that.
`ErrorText`	`String`	Gets or sets the text that can be displayed if an error is associated with the cell.
`FormattedValue`	`Object`	Gets the formatted version of the current value of the cell, after any formatting or type conversion has been applied.
`FormattedValue-Type`	`Type`	Gets the type that will be set on the cell for rendering, after formatting or type conversion has been applied.
`Frozen`	`Boolean`	True if either the row or column containing this cell is Frozen, false otherwise. (See the later section "Column and Row Freezing" for details on the meaning of this value.) This is a read-only property.

continues

TABLE 6.1: DataGridViewCell Properties (Continued)

Property Name	Type	Description
HasStyle	Boolean	True if a style has been explicitly set for this cell, false otherwise. This is a read-only property.
InheritedState	DataGridViewElementState	Gets the state enumeration that describes the states provided by the cell base class.
InheritedStyle	DataGridViewCellStyle	Gets the style that will be applied based on the styles of the grid, row, column, and default cell style. (See the later section on styles for details.)
IsInEditMode	Boolean	True if the cell is being edited by the user, false otherwise. This is a read-only property.
OwningColumn	DataGridViewColumn	Gets a reference to the cell's column.
OwningRow	DataGridViewRow	Gets a reference to the cell's row.
PreferredSize	Size	Gets the preferred size of the cell based on the cell template, which can be used in custom cell painting.
ReadOnly	Boolean	True if the contents of this cell are editable, false otherwise. This is a read/write property.
Resizable	Boolean	True if either the containing row or column is resizable, false otherwise.
RowIndex	Int32	Gets the index of the containing row within the grid.
Selected	Boolean	True if the cell is rendered as being selected and is marked as part of the selected cells collection, false otherwise. This is a read/write property.

TABLE 6.1: DataGridViewCell Properties (Continued)

Property Name	Type	Description
`Size`	`Size`	Gets the size of the entire cell.
`Style`	`DataGridViewCellStyle`	Gets or sets the Style object used for rendering this cell. (See the later section on styles for details on this property.)
`Tag`	`Object`	This simple placeholder reference, like other Windows Forms controls, lets you get or set an associated object reference for the cell. Typically, the tag is used as a place to stash a unique identifier for the control that can be used in lookup scenarios, such as looping through cells.
`ToolTipText`	`String`	Gets or sets the text that is rendered when the mouse hovers over a cell.
`Value`	`Object`	Probably the most important property on any cell. This property lets you get or set the value that the cell renders when it is displayed, but formatting and type conversion may occur automatically if the value set is different from the expected type of the cell.
`ValueType`	`Type`	Gets or sets the type of the value that is set against this cell, before formatting or type conversion applies. If the type hasn't been explicitly set, it is derived from the type of the containing column.
`Visible`	`Boolean`	True if both the containing row and column are visible, false otherwise. This is a read-only property.

The `DataGridViewColumn` (discussed earlier in this chapter) is the base class from which built-in column types derive. This class also has a number of useful properties that you can set to drive the behavior of the grid and that the type-specific column classes inherit. These properties are described in Table 6.2.

TABLE 6.2: DataGridViewColumn Properties

Name	Type	Description
`AutoSizeMode`	`DataGridView-AutoSizeColumnMode`	Gets or sets the autosizing behavior of the column (described in the later section on autosizing columns).
`CellTemplate`	`DataGridViewCell`	Gets or sets the cell type that will be used as a template for new cells that are created. Derived column types should limit the setting of a cell type to only the cell type they were designed to contain.
`CellType`	`Type`	Gets the type of cells this column was designed to contain.
`ContextMenuStrip`	`ContextMenuStrip`	Gets or sets the context menu object that will be presented when the user right-clicks in the column.
`DataPropertyName`	`String`	Gets or sets the property's name on bound data that will be used to set the value when data binding occurs.
`DefaultCellStyle`	`DataGridView-CellStyle`	Gets or sets the cell style that will be used by default for rendering the column's cells.

TABLE 6.2: DataGridViewColumn Properties (Continued)

Name	Type	Description
DisplayIndex	Int32	Gets or sets the display index of the column within the grid. This can be different than the ColumnIndex when column reordering is enabled (described later in this chapter).
DividerWidth	Int32	Gets or sets the width, in pixels, of the divider line that is drawn between this column and the next one to the right.
FillWeight	Float	Gets or sets the value used to determine the width of the column when in AutoSize-Mode of Fill.
Frozen	Boolean	True if the column is frozen, false otherwise. (Freezing columns and rows is discussed later in this chapter.) This is a read/write property.
HeaderCell	DataGridViewCell	Gets or sets the header cell (rendered at the top of the column).
HeaderText	String	Gets or sets the text that is rendered in the header cell as its value.
InheritedAuto-SizeMode	DataGridViewAuto-SizeColumnMode	Gets the autosize mode that is set for the base column class.
InheritedStyle	DataGridView-CellStyle	Gets the style that is inherited from the grid that will be used if none is assigned at the column, row, or cell level.

continues

TABLE 6.2: DataGridViewColumn Properties (Continued)

Name	Type	Description
`IsDataBound`	`Boolean`	True if the grid is operating in data-bound mode (a data source has been set), false otherwise. This is a read-only property.
`MinimumWidth`	`Int32`	Gets or sets the number of pixels used for a minimum width. This restricts the resizing of the column at runtime to be no less than this number.
`Name`	`String`	Gets or sets the name of the column. This is used for indexing into the columns collection on the grid and for data-binding purposes.
`ReadOnly`	`Boolean`	True if the cells in the column can be edited, false otherwise. This is a read/write property.
`Resizable`	`Boolean`	True if runtime resizing of the column is allowed by the user, false otherwise. This is a read/write property.
`Site`	`ISite`	Gets the `ISite` interface reference for the column, if any. This is used when the column is hosting a component.
`SortMode`	`DataGridView-ColumnSortMode`	Gets or sets the sort mode used for sorting the rows of the grid based on this column. This enumerated value can be set to `NotSortable`, `Automatic`, or `Programmatic`.

TABLE 6.2: DataGridViewColumn Properties (Continued)

Name	Type	Description
ToolTipText	String	Gets or sets the text that is shown as a tooltip pop-up when the mouse hovers over a cell in the column. If this property is set at the cell level, the value set for the cell is the one displayed.
ValueType	Type	Gets or sets the type of the values stored in the cells of this column type.
Visible	Boolean	True if the column will be displayed in the grid, false otherwise. This is a read/write property.
Width	Int32	Gets or sets the width, in pixels, of the column in the grid.

There are a number of built-in column types that are available for using with the `DataGridView` control corresponding to the most common control types that developers want to include in a grid. The following subsections describe each of the built-in column types and what is involved in using them.

DataGridViewTextBoxColumn

This is the default type of column (as described earlier in this chapter), and it displays text within the contained cells, which are of type `DataGridViewTextBoxCell`. Data that is bound to this column type and values set on the cell have to be of a type that can be converted to a string.

This column type supports editing if the `ReadOnly` property is `true` (the default) and the focus in on the cell. To enter editing mode, press F2, type in characters, or click in the cell. This embeds a separate editing control of type `DataGridViewTextBoxEditingControl`, which derives from

`TextBox`. This type enables in-place editing for the grid value, like you are used to for text box controls. The value in the text box is treated as a transient value until the focus leaves the cell; then the `CellParsing` event fires, and the value is pushed into the underlying data store if data bound or the `CellValuePushed` event fires if in virtual mode.

DataGridViewButtonColumn

This column type displays cells of type `DataGridViewButtonCell`, which is sort of a fancy form of read-only text cell. A button cell lets you have a button-push experience embedded in the grid, which you can use to trigger whatever action makes sense for your application. The button cell renders itself with a border that looks like any other button control, and when the user clicks on it, the cell renders again with a depressed offset so that you get an action like a button. To handle the "button click," you need to handle the `CellClick` event on the grid, determine if it was a button cell that was clicked, and then take the appropriate action for your application. This involves taking the event argument from the `CellClick` event, checking its `ColumnIndex` property against the column index of button columns in your grid, and then calling the button click handling code from there based on the row index, or the contents of that cell or others in that row.

DataGridViewLinkColumn

Like the button column, this is another form of rendering a text cell that gives the user a visual cue that clicking on it will invoke some action. This column type contains cells of type `DataGridViewLinkCell` and renders the text in the cell to look like a hyperlink. Typically, clicking on a link "navigates" the user somewhere else, so you might use this kind of column if you are going to pop up another window or modify the contents of another control based on the user clicking on the link. To do so, you handle the `CellClick` event as described previously for the button, determine if you are in a cell containing a link, and take the appropriate action based on that link. You will have to derive the context of what action you should take either from the cell's contents or other cells in that row or column.

DataGridViewCheckBoxColumn

By now you are probably picking up the pattern, and as you would guess, this column type contains cells of type `DataGridViewCheckBoxCell`. This cell type renders a `CheckBox`-like control that supports tri-state rendering like a `CheckBox` control.

The values that this cell type supports depend on whether you set the cell or column type into `ThreeState` mode or not. If the `ThreeState` property is set to `false` (the default), then a value of `null` or `false` will leave the check box unchecked; a value of `true` will check the box. If `ThreeState` is set to `true`, then the `Value` property of the cell can be `null` or one of the `CheckState` enumeration values. If `null` and `ThreeState` is `true`, then the check box will be rendered in the indeterminate state (a square filling it). The `CheckState` enumeration values are `Unchecked`, `Checked`, and `Indeterminate`, which are self-explanatory. The cell's `Value` property can be set explicitly through programmatic code that accesses the cell in the `Cells` collection of the row, or it can be set through data binding.

DataGridViewImageColumn

This column, not surprisingly, contains cells of type `DataGridViewImageCell`, which support the rendering of images directly within the grid's cells. This cell type provides a very handy and easy-to-use capability in the `DataGridView` control that used to be fairly painful to achieve with the `DataGrid` control. This column type exposes `Image` and `ImageLayout` properties in addition to the usual base class properties. Setting the column's `Image` property results in that image being displayed by default for all the cells in that column. The `ImageLayout` property takes a `DataGridViewImageCellLayout` enumeration value. The values of this enumeration and their effect on the rendering of an image in the grid are described in Table 6.3.

In addition to setting a default image at the column level, you can set the `Value` property at the cell level, either explicitly through code or implicitly through data binding. The value can be set to any type that can be converted to an `Image` object for display in the cell. Natively in .NET, this means that the value can either be an `Image` or a `byte` array that contains a serialized `Image`.

TABLE 6.3: DataGridViewImageCellLayout Enumeration Values and Effects

Value	Effect
`NotSet`	This is the default and indicates that the layout behavior has not been explicitly specified. The resulting behavior of the cell is the same as if `Normal` had been explicitly set.
`Normal`	The image is rendered at its native size and centered in the cell. Depending on the size of the cell, any portions of the image that are outside the bounds of the cell will be clipped.
`Stretch`	The image is stretched or shrunk in both width and height so that it fills the cell and no clipping occurs. No attempt is made to maintain the aspect ratio (width/height) of the image.
`Zoom`	The image is resized so that it fits within the cell without clipping, and the aspect ratio (width/height) is maintained so that no distortion of the image occurs.

DataGridViewComboBoxColumn

This column type contains cells of type `DataGridViewComboBoxCell`, which renders itself like a standard `ComboBox` control within the cell. This column type is definitely the most complex built-in column type for the `DataGridView`, and it exposes a number of properties that drive its behavior, as described in Table 6.4.

TABLE 6.4: DataGridViewComboBoxColumn Properties

Name	Type	Description
`AutoComplete`	`Boolean`	True if `AutoComplete` functionality is enabled when the cell is in edit mode, false otherwise. This lets users type in characters, and the combo box will select matching items in the list based on the characters typed. This is a read/write property.
`CellTemplate`	`DataGridViewCell`	Gets or sets the cell type being presented in the column. The cell type must be a derived class from `DataGridViewComboboxCell`.

TABLE 6.4: DataGridViewComboBoxColumn Properties (Continued)

Name	Type	Description
DataSource	Object	Gets or sets the object being used as the data source for data binding the column. Setting this property to a data collection has the same data-binding effect as it does with a normal combo box—it will display items from the collection as the items in the drop-down list, using the DisplayMember property to determine which data member or property within the items in that collection to use for the text in the list.
DisplayMember	String	Gets or sets which data member or property within the data source to display as the text items in the list.
DisplayStyle	DataGridViewComboBoxDisplayStyle	Gets or sets the style that the combo boxes in the column are using. The values for this enumeration include ComboBox, DropDownButton, and Nothing.
DisplayStyleForCurrentCellOnly	Boolean	True if the DisplayStyle value only applies to the current cell in the column, false if it is being used for all cells within the column. This is a read/write property.
DropDownWidth	Int32	Gets or sets the width of the drop-down list that is displayed when the user clicks on the down arrow or presses F4.
FlatStyle	FlatStyle	Gets or sets the FlatStyle enumerated value that determines the visual appearance of the combo box when it is rendered.

continues

TABLE 6.4: DataGridViewComboBoxColumn Properties (Continued)

Name	Type	Description
`Items`	`ObjectCollection`	Gets the collection of objects that are set for the cell template.
`MaxDropDownItems`	`Int32`	Gets or sets the maximum number of items to display in the drop-down list.
`Sorted`	`Boolean`	True if the items in the list will be sorted alphabetically, false otherwise. This is a read/write property.
`ValueMember`	`String`	Gets or sets the data member or property within the data source that will be kept with the items in the list. Lets you keep track of additional information, such as the record primary key.

The combo box cells support edit mode, and users can type in a value for autocompletion purposes or select values from a drop-down list. When in edit mode, this cell type hosts a control that derives from the `ComboBox` control, so all of its functionality is exposed when the cell is switched into edit mode.

The `Value` property represents the currently selected value in the combo box. It may contain the displayed text value in the combo box, or it may contain the underlying `ValueMember` value for the selected item, depending on what you set for the `DataSource`, `DisplayMember`, and `ValueMember` properties. The `FormattedValue` property, inherited from the base class, always contains the formatted text for the selected item that is being displayed in the combo box.

Data binding this column type or the cells in it works just like data binding a standalone `ComboBox` control. You set the `DataSource`, `DisplayMember`, and `ValueMember` properties, and the items in the data source collection are rendered in the drop-down list using the value of the data member that is identified as the display member:

```
toCountryColumn.DataSource = m_CountriesBindingSource;
toCountryColumn.DisplayMember = "CountryName";
toCountryColumn.ValueMember = "CountryID";
```

The sample code that accompanies this chapter contains a simple application called ColumnTypes that demonstrates how the code interacts with each of the built-in column types described in this chapter.

Built-In Header Cells

Header cells are the cells rendered at the top and left sides of the grid. They provide the context or a guide to what the cells in the grid contain. Column header cells are of type `DataGridViewColumnHeaderCell`, and their header text indicates the content of the column cells. The cell contains an up or down triangle when the column supports sorting; users can sort the column by clicking on the column header. Usually the header text is set through the `HeaderText` property on the column, either explicitly through code or implicitly through data binding based on the data's schema. You can also access the header cell directly from the column through the `HeaderCell` property and use its `Value` to set the text displayed.

The row header cells are of type `DataGridViewRowHeaderCell`. They indicate row selections with a triangle glyph, editing mode with a pencil glyph, and the new row with a star glyph. Row header cells can display text as well; you set the cell's `Value` to a string value by accessing the row's `HeaderCell` property.

Both column and row headers can be further customized by implementing custom painting by handling the `CellPainting` event on the grid. Note that if you do custom painting, you must do all the painting of the header cell yourself, and then set the `Handled` property on the event argument to true:

```
private void OnCellPainting(object sender,
   DataGridViewCellPaintingEventArgs e)
{
   if (e.ColumnIndex < 0)
   {
      e.Graphics.FillRectangle(Brushes.Aqua, e.CellBounds);
      e.Handled = true;
   }
}
```

This code checks to see if the column being painted has an index less than zero, which indicates that the row header is being painted. The column index of the row headers is –1, and the row index of the column headers is also –1. You cannot index into the `Cells` collection on the row with these values, but you can use them as a flag in the `CellPainting` event to know when it is a header that is being painted.

Additionally, you can set the `CellHeader` property to an instance of a class that derives from `DataGridViewCell`, and then that cell type will be used when the header cells are rendered. You can derive your own class from the cell base class and do whatever kind of custom painting, formatting, or setting of styles there that makes sense.

Handling Grid Data Edits

How you handle grid edits is going to depend on the following:

- The type of column or cell you are dealing with
- Whether the data is data bound
- Whether you are in virtual mode

As mentioned earlier, when working with a text box column, users can start editing a cell by putting the focus into the cell with the mouse, arrow keys, or by pressing the F2 key when the mouse pointer is in the cell. If users then start typing characters, the current contents of the cell will be overwritten. When they change the focus to another cell, this completes the editing process.

The first thing that happens that you might want to handle is that the `CellParsing` event fires. Like its `CellFormatting` counterpart, this event gives you an opportunity to intercept the value that was entered into the cell while in edit mode, either to handle storing that value somewhere yourself or to transform it into some other value before it is stored.

If the cell is data bound, and if the data source supports editing the data objects in the collection, the data will automatically be pushed back

into the underlying data source. If the cell is a button or link cell, however, you won't be able to edit the contents in the first place because they don't support editing. If the cell is a combo box cell, editing is done by selecting a value in the drop-down list or overtyping the current selection if the cell has its `DisplayStyle` property set to `ComboBox`. This changes the cell's value when editing is complete (when the focus moves off the cell) and results in the same action as if that value had been typed into a text box cell. If the grid is in virtual mode, you will need to handle the `CellValue-Pushed` event to grab the value that was entered and do what you need to with it.

When a cell switches into edit mode, an event named `Editing-ControlShowing` fires. This event passes an event argument that lets you get a reference to the editing control itself. The built-in cell types that support editing (text box, combo box, and check box cell types) create an instance of an editing control that derives from their normal Windows Forms counterparts (`TextBox`, `ComboBox`, and `CheckBox`, respectively) and display that control as a child control inside a panel inside the cell. If you create a custom cell type that supports editing, then you will want to follow a similar approach. Through the `EditingControlShowing` event, you can get a reference to the editing control in use and can tap into its event model to respond to edits in realtime. For example, if you want to dynamically react to selections in a combo box column while the control is still in edit mode and the selected value hasn't been pushed yet into the cell's underlying value (meaning the `CellParsing` event hasn't yet fired), you could use the `EditingControlShowing` event to hook things up:

```
public Form1()
{
   InitializeComponent();
   m_Grid.EditingControlShowing += OnEditControlShowing();
}

private void OnEditControlShowing(object sender,
   DataGridViewEditingControlShowingEventArgs e)
{
   if (m_Grid.CurrentCell.ColumnIndex == 2)
   {
```

```
            m_HookedCombo = e.Control as ComboBox;
            if (m_HookedCombo == null)
               return;
            m_HookedCombo.SelectedIndexChanged += OnCountryComboChanged;
         }
      }

      void OnCountryComboChanged(object sender, EventArgs e)
      {
         string countryName =
            (string)m_Grid.CurrentCell.EditedFormattedValue;
         if (string.IsNullOrEmpty(countryName))
            return;
         DataRow[] countries = m_MoneyData.Countries.Select(
            string.Format("CountryName = '{0}'", countryName));
         if (countries != null && countries.Length > 0)
         {
            MoneyDBDataSet.CountriesRow row =
               countries[0] as MoneyDBDataSet.CountriesRow;
            int flagColIndex = m_Grid.Columns["TargetCountryFlag"].Index;
            DataGridViewCell cell = m_Grid.CurrentRow.Cells[flagColIndex];
            cell.Value = row.Flag;
         }
      }
```

This code does the following:

1. The constructor subscribes the `OnEditControlShowing` method to the grid's `EditControlShowing` event.
2. When the `EditControlShowing` event fires, the `OnEditControl-Showing` method uses the `Control` property on the event argument to get a reference to the `ComboBox` control that is embedded in the cell that is being edited.
3. The `OnEditControlShowing` method then subscribes the `OnCountryComboChanged` method to the `SelectedIndexChanged` event on that `ComboBox` control.
4. When the `SelectedIndexChanged` event fires, the `OnCountry-ComboChanged` method retrieves the country name from the cell containing the drop-down list using the current cell's `Edited-FormattedValue` property. This lets you get the edited value before the cell has left editing mode.

5. The `OnCountryComboChanged` method then uses the country name to retrieve the corresponding row in the Countries table and extracts the flag image from the Flag column.
6. Finally, it sets the flag image as the value on the cell corresponding to the country's flag.

Keep in mind that the Flag column in the Countries table is actually a byte array containing the bits of the saved image file. The automatic formatting of the image column kicks in here to present the image in the same way that was discussed for a `PictureBox` control in Chapter 4. The ColumnTypes sample in the download code demonstrates this technique.

Automatic Column Sizing

One of the `DataGridView` control's new features is its ability to automatically calculate the width of the columns to fit the content of the columns based on several different criteria. Like many of the grid's features, all you need to do to take advantage of this feature is to set the appropriate property on a given column—and then the grid does the rest. Specifically, the property that takes care of this for you is the `AutoSizeMode` property of the `DataGridViewColumn` class. By setting this property to one of the enumerated values of the `DataGridViewAutoSizeColumnMode` enumeration shown in Table 6.5, you can drive how to set the width of columns in the grid.

TABLE 6.5: DataGridView AutoSizeMode Values

Value	How the Column Width Is Calculated
`NotSet`	By the value set on the `AutoSizeColumnsMode` property at the grid level. This is the default value.
`None`	Set explicitly by setting the column's `Width` property.
`ColumnHeader`	By the width of the content in the header cell only.
`AllCellsExcept-Header`	By the width of the widest cell in the grid, whether it is displayed or not, but the header cell content size is ignored.

continues

TABLE 6.5: DataGridView AutoSizeMode Values (Continued)

Value	How the Column Width Is Calculated
`AllCells`	By the width of all cells in the column, including those not displayed and the header cell content.
`DisplayedCells-ExceptHeader`	Based only on displayed cells in the column, ignoring the width of the header cell content.
`DisplayedCells`	By the width of the content in displayed cells, including the header cell.
`Fill`	Automatically calculated to fill the displayed content area of the grid so the contents can be viewed without scrolling. The actual value used depends on the mode of the other columns in the grid, and their `MinimumWidth` and `FillWeight` properties. If all the columns in the grid are set to `Fill` mode, and their minimum width requirements are met, then the columns will each have equal width, filling the grid, but not requiring any horizontal scrolling.

One of the most useful values is `AllCells`. I recommend that this be your default, unless you see a performance hit from using it for large data sets or if you have some cell values that will be very long. This setting ensures that the content of cells never wraps. Additionally, remember to set the `FormattingApplied` property on the event argument to the `CellFormatting` event if you are dynamically populating cell values. Otherwise, setting the `AutoSizeMode` to one of the row values will result in an infinite loop.

As a simple example of using this feature, the following code modifies the code from Listing 6.1 to set the column width of the Full Name computed column:

```
newColIndex = m_AuthorsGrid.Columns.Add("FullName", "Full Name");
m_AuthorsGrid.Columns[newColIndex].AutoSizeMode =
   DataGridViewAutoSizeColumnMode.AllCells;
```

The `Fill` mode is very powerful for automatically maximizing the use of the grid real estate, but it can be a little complicated to understand. Basically, if you set the mode for all columns to `Fill`, each of the columns will have their width set equally, and the columns will fill the grid boundary

with no horizontal scrollbar needed. If the `MinimumWidth` property of any column set to `Fill` mode is wider than the width that was computed using the fill algorithm, then the `MinimumWidth` value will be used instead, and the other columns just end up being narrower so that they all still fit in the grid without a horizontal scrollbar. If the `MinimumWidth` values of multiple columns make it so that all columns cannot be displayed, then the columns that cannot be displayed will be set to their minimum width value, and a scrollbar will be displayed. The default value for minimum width is only 5 pixels, so you will definitely want to set a more sensible `MinimumWidth` value when working with `Fill` mode.

Each column also has a `FillWeight` property, which takes effect when that column's `AutoSizeMode` is set to `Fill`. The `FillWeight` can be thought of as the percentage of the remaining available grid width that the individual column will take up compared to other columns that are set to `Fill`. It is a weight instead of a percentage, though, because you can use values that don't add up to 100. For example, suppose you wanted to display the CustomerID, CompanyName, and ContactName columns from the Northwind Customers table in a grid. The following code sets the column width of the CustomerID column to 75 pixels, and then sets the remaining two columns to `Fill` mode with weights of 10 and 20, respectively.

```
public Form1()
{
   InitializeComponent();
   m_CustomersGrid.Columns["CustomerID"].Width = 75;
   m_CustomersGrid.Columns["CompanyName"].AutoSizeMode =
      DataGridViewAutoSizeColumnMode.Fill;
   m_CustomersGrid.Columns["CompanyName"].FillWeight = 10;
   m_CustomersGrid.Columns["ContactName"].AutoSizeMode =
      DataGridViewAutoSizeColumnMode.Fill;
   m_CustomersGrid.Columns["ContactName"].FillWeight = 20;
}
```

As a result, the remaining two columns occupy 33 percent and 67 percent of the remaining grid width, respectively, after the CustomerID column has taken up its fixed width space. Figure 6.2 illustrates this.

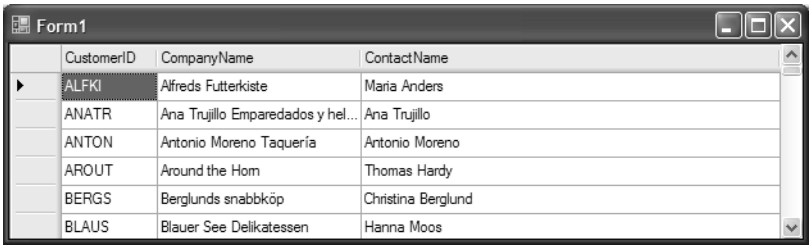

FIGURE 6.2: Columns Using AutoSizeMode of Fill and FillWeights

Column and Row Freezing

Scrolling is inevitable when dealing with lots of rows or columns of data. Often when you scroll through data, it is easy to lose the context of what rows or columns you are looking at, especially if that context is based on the values in some other rows or columns. Let's say you are browsing through a grid filled with product information. If there are a lot of columns of data associated with each product, as you scroll to the right to view columns that aren't currently displayed, you will lose the context of the product name as it gets scrolled off the left of the screen. What you would really want in this situation is to be able to freeze the product name column so that it is always shown and only have the remaining columns scroll. Likewise, there may be situations where you need to present one or more rows at the top of the grid that need to remain in place as you scroll down to additional rows in the grid.

Accomplishing this with the `DataGridView` control is simple: You just set the `Frozen` property to `true` on any row or column to get this behavior. Specifically, if you freeze a column, then that column, and all the columns to the left of it, won't scroll when you scroll to the right in the grid. Similarly, if you freeze a row, then that row and all the rows above it won't scroll when you scroll down in the grid. If you are going to freeze a column or row, then you will probably want to provide a visual cue to the user indicating the logical boundary that exists between the frozen item and the nonfrozen ones next to it. The easiest way to do this is to set the `DividerWidth` property on the column or row to something other than the default. This property is an integer that specifies the number of pixels used to draw

the divider between cells of that column or row and the adjacent one (to the right or below).

The following code shows a simple example of freezing both a column and a row and setting the divider width:

```
m_ProductsGrid.Columns["ProductName"].Frozen = true;
m_ProductsGrid.Columns["ProductName"].DividerWidth = 3;
m_ProductsGrid.Rows[1].Frozen = true;
m_ProductsGrid.Rows[1].DividerHeight = 3;
```

Using the Designer to Define Grids

Now that you understand how to code most of the common uses of the grid, let's cover how to avoid having to write a lot of that code yourself. The `DataGridView` control supports a very rich experience while working in the Visual Studio designer through a combination of the designer Smart Tags, dialogs, and the Properties window.

For starters, if you have defined a data source in your project, you can simply drag a data collection source like a data table onto the form designer and a `DataGridView` instance will be created with all of its supporting objects. Additionally, the column definitions based on the grid's data source properties let you set other properties, such as the `AutoSizeMode`, using the designer. If you select the grid and display its Smart Tag, as shown in Figure 6.3, you can modify the most common options of the grid's appearance and behavior from there.

The *Choose Data Source* drop-down displays a data sources selection window similar to the one described in Chapter 5 for the Properties window. The presented data sources will be tailored to only those that implement the `IList` interface and thus are suitable for binding to the grid.

The *Edit Columns* and *Add Column* links display dialogs that let you define the columns that the grid will contain, shown in Figures 6.4 and 6.5 respectively.

The Edit Columns dialog lets you add and remove columns, set the order of the columns within the grid, and set all the design-time properties for a defined column in a focused dialog. The properties shown in the dialog will be tailored based on whether the column is a bound or unbound

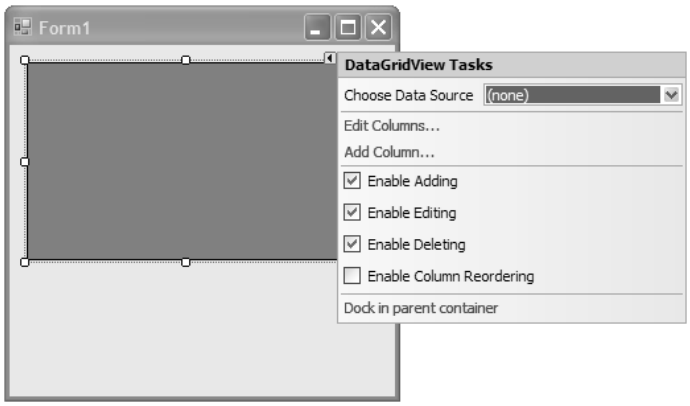

FIGURE 6.3: DataGridView Smart Tag

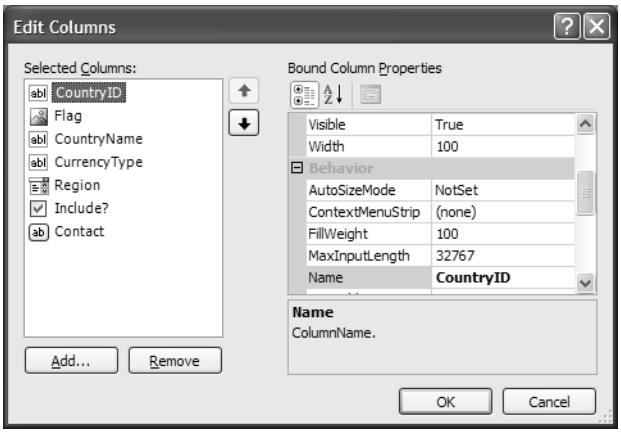

FIGURE 6.4: Edit Columns Dialog

column, and will expose additional properties based on the column type and cell type. If you define custom column types and include them in your project, they will show up as options for new columns or for configuring columns through this dialog.

The Add Column dialog (see Figure 6.5) lets you add a new data-bound or unbound column to the grid. If you are adding a data-bound column, you can select from the columns available in the currently selected data source. You will first have to set the data source to an appropriate collection of data either through the Smart Tag or through the `DataSource` property in the Properties window. If you are adding an unbound column,

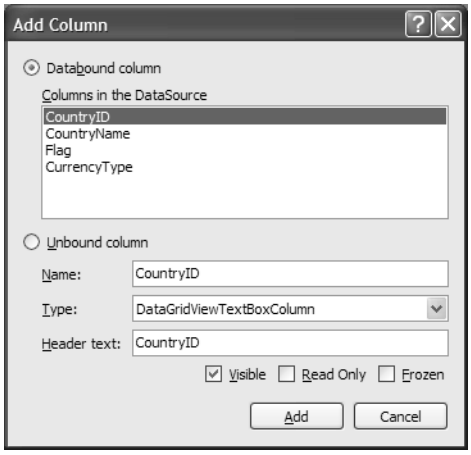

FIGURE 6.5: Add Column Dialog

then you just specify the name of the column, the type of the column, and the header text. When you click the *Add* button, the column is added to the grid, and the dialog remains open so you can quickly define multiple new columns.

Configuring the columns through these dialogs writes all the code for you that has been covered earlier in this chapter for defining columns and controlling their runtime behavior.

The *Enable Adding* check box on the `DataGridView` Smart Tag sets the `AllowUserToAddRows` property to `true` if checked, which displays a new empty row at the bottom of the grid. This lets users add a new row to the data collection by typing new values into the cells. The ability to support this depends on whether the grid is data bound, and, if so, whether the underlying object collection supports adding new items to the collection (see the discussion in Chapter 7). Likewise, the *Enable Editing* check box sets the `ReadOnly` property, which affects whether users can edit the contents of the grid in place, and *Enable Deleting* sets the `AllowUserToDeleteRows` property. The *Enable Column Reordering* check box sets the `AllowUser-ToOrderColumns` property, whose behavior is described in the next section.

The *Dock in parent container* link is only available if you first drag and drop a grid control onto a form. It does exactly what it says—it simply sets the `Dock` property to `Fill`.

In addition to the common properties and behaviors that you can configure through the Smart Tag, there are a bunch of other properties and events that you can configure at design time through the Properties window. Setting any of these properties generates appropriate code in the designer-generated partial class for the form in the `InitializeComponent` method. Most notably, you can configure any of the data-binding properties through the Properties window. You'll probably want to set styles using the Properties window, because you can preview the results of those settings in the designer to make sure you are getting what you expect. Styles are discussed in more detail at the end of this chapter.

Column Reordering

Column reordering is a slick built-in behavior of the grid that lets users change the display order of columns in the grid at runtime. Because different users of an application often pay more attention to some columns in a grid than others, users commonly request to set the order of the columns displayed in the grid themselves. While you could support this functionality by programmatically removing columns from the grid and then inserting them back in the new position, that requires a fair amount of tedious code to have to write for a common use case. So the Windows Client team was nice enough to build functionality for this right into the grid control.

The way this works is that if the `AllowUserToOrderColumns` property is set to `true` and the user clicks and drags on a column header, the grid lets them drag and drop the column to the position where they would like it to display. The columns to the right of the drop position will move one position to the right, and the columns surrounding the original location of the dragged column will move to be adjacent after the column has been moved. Figure 6.6 shows this in action. In this case, the QuantityPerUnit column was clicked on and is being dragged to the left. A gray box is drawn the size of the column's header cell you are dragging. When you move the cursor to one side of another column, the border between that column and the adjacent one darkens, indicating where the column you are dragging will be placed if you release the mouse button.

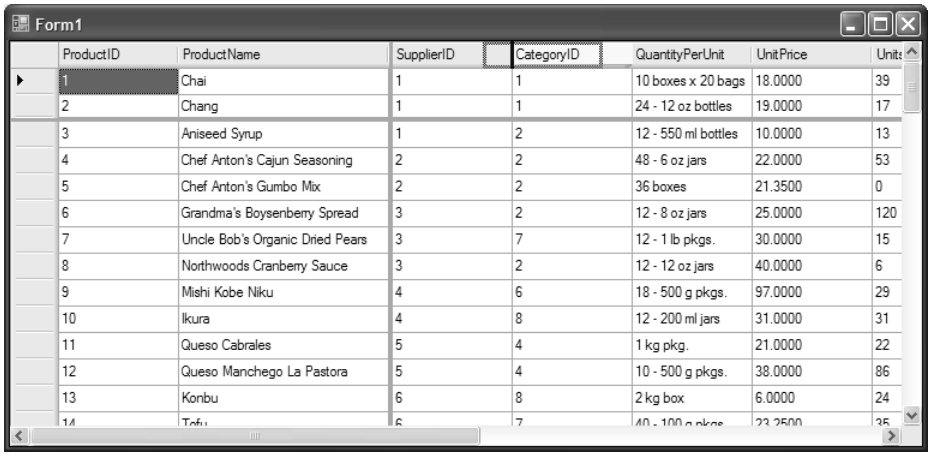

FIGURE 6.6: Column Reordering in Action

When a column has been moved through column reordering, its `ColumnIndex` doesn't change, but the `DisplayIndex` property indicates its current display order within the grid. By default, the display order of the grid is not persisted between application runs, but it's a simple matter to persist that information yourself and reinstate the display order by writing the display order to a file. The code in Listing 6.3 demonstrates persisting the data into a file in isolated storage using the `XmlSerializer` class.

LISTING 6.3: Persisting Display Order of Columns

```
public partial class Form1 : Form
{
   public Form1()
   {
      InitializeComponent();
   }


   private void Form1_Load(object sender, EventArgs e)
   {
      m_Grid.AllowUserToOrderColumns = true;
      SetDisplayOrder();
   }

   private void OnFormClosing(object sender, FormClosingEventArgs e)
   {
      CacheDisplayOrder();
   }
```

continues

```
    private void CacheDisplayOrder()
    {
       IsolatedStorageFile isoFile =
          IsolatedStorageFile.GetUserStoreForAssembly();
       using (IsolatedStorageFileStream isoStream = new
          IsolatedStorageFileStream("DisplayCache", FileMode.Create,
             isoFile))
       {
          int[] displayIndices =new int[m_Grid.ColumnCount];
          for (int i = 0; i < m_Grid.ColumnCount; i++)
          {
             displayIndices[i] = m_Grid.Columns[i].DisplayIndex;
          }
          XmlSerializer ser = new XmlSerializer(typeof(int[]));
          ser.Serialize(isoStream,displayIndices);
       }
    }

    private void SetDisplayOrder()
    {
       IsolatedStorageFile isoFile =
          IsolatedStorageFile.GetUserStoreForAssembly();
       string[] fileNames = isoFile.GetFileNames("*");
       bool found = false;
       foreach (string fileName in fileNames)
       {
          if (fileName == "DisplayCache")
             found = true;
       }
       if (!found)
          return;
       using (IsolatedStorageFileStream isoStream = new
          IsolatedStorageFileStream("DisplayCache", FileMode.Open,
             isoFile))
       {
          try
          {
             XmlSerializer ser = new XmlSerializer(typeof(int[]));
             int[] displayIndicies =
                (int[])ser.Deserialize(isoStream);
             for (int i = 0; i < displayIndicies.Length; i++)
             {
                m_Grid.Columns[i].DisplayIndex = displayIndicies[i];
             }
          }
          catch { }
       }
    }
}
```

This code isn't specific to the data source in any way. The key facets here are that the code in the form `Load` event handler sets the `AllowUserToOrderColumns` property to `true`, allowing the dynamic changing of `DisplayIndex` for columns through drag-and-drop operations. I then added a `CacheDisplayOrder` helper method that is called by the `Form.Closing` event handler, and a `SetDisplayOrder` helper method that is called when the form loads.

`CacheDisplayOrder` first collects all the display index values for each of the grid's columns and puts them into an integer array. It then creates an isolated storage file stream and writes the array to that stream using the `XmlSerializer` class. The `SetDisplayOrder` method does the reverse: it first checks to see if the file exists, and if so, reads the array back in and uses it to set the `DisplayIndex` on each column in the grid.

Defining Custom Column and Cell Types

With the `DataGridView`, you are already leaps and bounds ahead of the `DataGrid` for presenting rich data because of the built-in column types that it supports out of the box. But there are always custom scenarios that you will want to support to display custom columns. Luckily, another thing the `DataGridView` makes significantly easier is plugging in custom column and cell types.

If you want to customize just the painting process of a cell, but you don't need to add any properties or control things at the column level, you have an event-based option rather than creating new column and cell types. You can handle the `CellPainting` event and draw directly into the cell itself, and you can achieve pretty much whatever you want with the built-in cell types and some (possibly complex) drawing code. But if you want to be able to just plug your column or cell type in a reusable way with the same ease as using the built-in types, then you can derive your own column and cell types instead.

The model you should follow for plugging in custom column types matches what you have already seen for the built-in types: You need to create a column type and a corresponding cell type that the column will contain. You do this by simply inheriting from the base `DataGridViewColumn`

and `DataGridViewCell` classes, either directly or indirectly, through one of the built-in types.

The best way to explain this in detail is with an example. Say I wanted to implement a custom column type that lets me display the status of the items represented by the grid's rows. I want to be able to set a status using a custom-enumerated value, and cells in the column will display a graphic indicating that status based on the enumerated value set on the cell. To do this, I define a `StatusColumn` class and a `StatusCell` class (I disposed of the built-in type naming convention here of prefixing `DataGridView` on all the types because the type names get sooooooooooo long). I want these types to let me simply set the value of a cell, either programmatically or through data binding, to one of the values of a custom-enumerated type that I call `StatusImage`. `StatusImage` can take the values `Green`, `Yellow`, or `Red`, and I want the cell to display a custom graphic for each of those based on the value of the cell. Figure 6.7 shows the running sample application with this behavior.

Defining a Custom Cell Type

To achieve this, the first step is to define the custom cell type. If you are going to do your own drawing, you can override the protected virtual `Paint` method from the `DataGridViewCell` base class. However, if the cell content you want to present is just a variation on one of the built-in cell types, you should consider inheriting from one of them instead. That is what I did in this case. Because my custom cells are still going to be presenting images, the `DataGridViewImageCell` type makes a natural base class. My `StatusCell` class isn't going to expose the ability to set the image at random, though; it is designed to work with enumerated values.

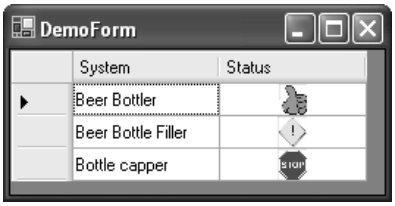

FIGURE 6.7: Custom Column and Cell Type Example

I also want the cell value to be able to handle integers as long as they are within the corresponding numeric values of the enumeration, so that I can support the common situation where enumerated types are stored in a database as their corresponding integer values. The code in Listing 6.4 shows the `StatusCell` class implementation.

LISTING 6.4: Custom Cell Class

```
namespace CustomColumnAndCell
{
   public enum StatusImage
   {
      Green,
      Yellow,
      Red
   }

   public class StatusCell : DataGridViewImageCell
   {
      public StatusCell()
      {
         this.ImageLayout = DataGridViewImageCellLayout.Zoom;
      }

      protected override object GetFormattedValue(object value,
         int rowIndex, ref DataGridViewCellStyle cellStyle,
         TypeConverter valueTypeConverter,
         TypeConverter formattedValueTypeConverter,
         DataGridViewDataErrorContexts context)
      {
         string resource = "CustomColumnAndCell.Red.bmp";
         StatusImage status = StatusImage.Red;
         // Try to get the default value from the containing column
         StatusColumn owningCol = OwningColumn as StatusColumn;
         if (owningCol != null)
         {
            status = owningCol.DefaultStatus;
         }
         if (value is StatusImage || value is int)
         {
            status = (StatusImage)value;
         }
         switch (status)
         {
            case StatusImage.Green:
               resource = "CustomColumnAndCell.Green.bmp";
               break;
```

continues

```
                case StatusImage.Yellow:
                    resource = "CustomColumnAndCell.Yellow.bmp";
                    break;
                case StatusImage.Red:
                    resource = "CustomColumnAndCell.Red.bmp";
                    break;
                default:
                    break;
            }
            Assembly loadedAssembly = Assembly.GetExecutingAssembly();
            Stream stream =
                loadedAssembly.GetManifestResourceStream(resource);
            Image img = Image.FromStream(stream);
            cellStyle.Alignment =
                DataGridViewContentAlignment.TopCenter;
            return img;
        }
    }
}
```

The first declaration in this code is the enumeration `StatusImage`. That is the value type expected by this cell type as its `Value` property. You can then see that the `StatusCell` type derives from the `DataGridViewImageCell`, so I can reuse its ability to render images within the grid. There is a default status field and corresponding property that lets the default value surface directly. The constructor also sets the `ImageLayout` property of the base class to `Zoom`, so the images are resized to fit the cell with no distortion.

The key thing a custom cell type needs to do is either override the `Paint` method, as mentioned earlier, or override the `GetFormattedValue` method as the `StatusCell` class does. This method will be called whenever the cell is rendered and lets you handle transformations from other types to the expected type of the cell. The way I have chosen to code `GetFormattedValue` for this example is to first set the value to a default value that will be used if all else fails. The code then tries to obtain the real default value from the containing column's `DefaultValue` property if that column type is `StatusColumn` (discussed next). The code then checks to see if the current `Value` property is a `StatusImage` enumerated type or an integer, and if it is an integer, it casts the value to the enumerated type.

Once the status value to be rendered is determined, the `GetFormattedValue` method uses a switch-case statement to select the appropriate

resource name corresponding to the image for that status value. You embed bitmap resources in the assembly by adding them to the Visual Studio project and setting the Build Action property on the file to *Embedded Resource*. The code then uses the `GetManifestResourceStream` method on the `Assembly` class to extract the bitmap resource out of the assembly, sets the alignment on the `cellStyle` argument passed into the method, and then returns the constructed image as the object from the method. The object that you return from this method will be the one that is passed downstream to the `Paint` method as the formatted value to be rendered. Because this doesn't override the `Paint` method, the implementation of my `DataGridViewImageCell` base class will be called, and it expects an `Image` value to render.

Defining a Custom Column Type

So now you have a custom cell class that could be used in the grid, but you also want to have a custom column class that contains `StatusCells` and can be used for setting up the grid and data binding. If you were going to use the custom cell type completely programmatically, you could just construct an instance of the `DataGridViewColumn` base class and pass in an instance of a `StatusCell` to the constructor, which sets that as the `CellTemplate` for the column. However, that approach wouldn't let you use the designer column editors covered in Figures 6.4 and 6.5 to specify a bound or unbound column of `StatusCells`. To support that, you need to implement a custom column type that the designer can recognize. As long as you're implementing your own column type, you also want to expose a way to set what the default value of the `StatusImage` should be for new rows that are added. The implementation of the `StatusColumn` class is shown in Listing 6.5.

LISTING 6.5: Custom Column Class

```
namespace CustomColumnAndCell
{
   public class StatusColumn : DataGridViewColumn
   {
      public StatusColumn() : base(new StatusCell())
      {
      }
```

continues

```
        private StatusImage m_DefaultStatus = StatusImage.Red;

        public StatusImage DefaultStatus
        {
           get { return m_DefaultStatus; }
           set { m_DefaultStatus = value; }
        }

        public override object Clone()
        {
           StatusColumn col = base.Clone() as StatusColumn;
           col.DefaultStatus = m_DefaultStatus;
           return col;
        }

        public override DataGridViewCell CellTemplate
        {
           get { return base.CellTemplate; }
           set
           {
              if ((value == null) || !(value is StatusCell))
              {
                 throw new ArgumentException(
"Invalid cell type, StatusColumns can only contain StatusCells");
              }
           }
        }
     }
}
```

You can see from the implementation of `StatusColumn` that you first need to derive from the `DataGridViewColumn` class. You implement a default constructor that passes an instance of your custom cell class to the base class constructor. This sets the `CellTemplate` property on the base class to that cell type, making it the cell type for any rows that are added to a grid containing your column type.

The next thing the class does is define a public property named `DefaultStatus`. This lets anyone using this column type to set which of the three `StatusImage` values should be displayed by default if no value is explicitly set on the grid through data binding or programmatic value setting on a cell. The setter for this property changes the member variable that keeps track of the current default. The `DefaultStatus` property on the

column is accessed from the `StatusCell.GetFormattedValue` method, as described earlier.

Another important thing for you to do in your custom column type is to override the `Clone` method from the base class, and in your override, return a new copy of your column with all of its properties set to the same values as the current column instance. This method is used by the design column editors to add and edit columns in a grid through the dialogs discussed in Figures 6.4 and 6.5.

The last thing the custom column class does is to override the `CellTemplate` property. If someone tries to access the `CellTemplate`, the code gets it from the base class. But if someone tries to change the `CellTemplate`, the setter checks to see if the type of the cell being set is a `StatusCell`. If not, it raises an exception, preventing anyone from programmatically setting an inappropriate cell type for this column. This doesn't prevent you from mixing other cell types into the column for a heterogeneous grid (as shown earlier in the section on programmatically creating the grid).

Now that you have defined the custom cell and column types, how can you use them? Well, you can define them as part of any Windows application project type in Visual Studio, but generally when you create something like this, you are doing it so you can reuse it in a variety of applications. Whenever you want reuse code, you need to put that code into a class library. So if you define a class library project, add the classes just discussed to the class library, along with the images you want to use for displaying status as embedded resources in the project. This creates an assembly that you can then reference from any Windows application that you want to use the column and cell type within. All you need to do is set a reference to that assembly from the Windows Forms project in which you want to use them, and the custom column types will display in the Add Column dialog, as shown in Figure 6.8 (`StatusColumn`, in this case).

Within your Windows Forms application, you can programmatically add `StatusColumns` to a grid, or use the designer to do so. If you add the column through the designer and then look at it in the Edit Columns dialog, you will see that `DefaultStatus` appears in the property list and is

settable as an enumerated property with its allowable values (see Figure 6.9).

With a column of this type added to the grid, you can either populate the grid programmatically with either of the types that the cell is able to handle for values (either `StatusImage` values or integers within the value range of `StatusImage`), or you can data bind to it with a collection of data that contains those values. Here is a simple example of setting the values programmatically on a grid containing two columns: a text box column

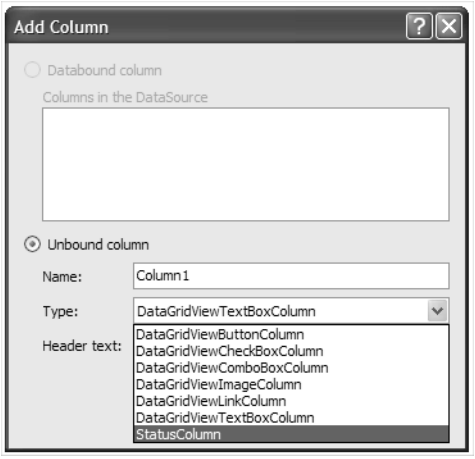

FIGURE 6.8: Custom Column Types in the Add Column Dialog

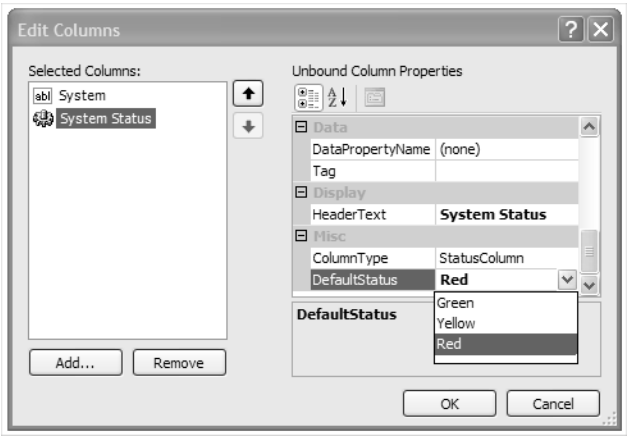

FIGURE 6.9: Custom Column Properties in the Edit Columns Dialog

and a `StatusColumn`. Note that you can set the values with either the enumerated value or with an appropriate integer value.

```
m_Grid.Rows.Add("Beer Bottler", StatusImage.Green);
m_Grid.Rows.Add("Beer Bottle Filler", 1); //StatusImage.Yellow = 1
m_Grid.Rows.Add("Bottle capper", StatusImage.Red);
```

The CustomColumnAndCell sample application in the download code also demonstrates creating a data set and data binding against the status column.

Utilizing Cell-Oriented Grid Features

You have probably noticed that the `DataGridView` is much more focused at the cell level that its `DataGrid` predecessor was. Part of the reason for this is that a frequent use of grids is where columns don't necessarily dictate the structure of the grid's content. Specifically, users want spreadsheet-like functionality that mimics the interaction model millions of people have become accustomed to with programs like Microsoft Excel and other spreadsheet applications.

Once again, the `DataGridView` comes to the rescue and makes supporting that model fairly easy. You have already seen some of the cell-level events that let you control what is displayed at the cell level (`CellFormatting` event) and that tell you when users interact with a cell by editing the contents (`EditControlShowing` event) or simply click on it (`CellClick` event). You can set different context menus and tooltips down to the cell level, so that every cell can become a distinct entity in the grid from the users' perspective. There are actually over 30 events raised by the `DataGridView` that expose interactions and modifications at the cell level that you can subscribe to for providing cell-oriented features.

Additionally, there are different selection modes that you can use to change the way the grid highlights cells, columns, or rows when the user clicks in different places in the grid. The `SelectionMode` property on the grid determines the selection behavior and is of type `DataGridViewSelectionMode`. The `DataGridView` control supports the selection modes (described in Table 6.6). While you can't combine these modes (the

TABLE 6.6: DataGridViewSelectionMode Enumeration Values

Value	Description
`CellSelect`	This mode lets you select one or many cells in the grid using the mouse or keyboard. If you click in any cell, just that cell will be selected. You can click and drag, and contiguous cells that you drag over will also be selected. If you click in one cell, then Shift-click in another, you will select the entire contiguous set of cells from the first click to the second. You can even select noncontiguous cells by holding down the Ctrl key while you click cells. This is the default selection mode.
`FullRowSelect`	Clicking in any cell in the grid will select all of the cells in the entire row that contains the cell and will deselect any cells outside the row.
`FullColumnSelect`	Clicking in any cell in the grid will select all of the cells in the entire column that contains the cell and will deselect any cells outside the column.
`RowHeaderSelect`	Clicking on the row header cell will select the entire row, but otherwise this selection mode behaves like `CellSelect`. This is the mode set by the designer for a grid when you add it to a form.
`ColumnHeaderSelect`	Clicking on the column header cell will select the entire column, but otherwise this selection mode behaves like `CellSelect`.

enumeration isn't a Flags enumerated type), you can achieve a combination of modes by using the `SelectionMode` property on the grid plus some additional event handling. Regardless of which of these modes you select, clicking on the upper left header cell (the one that is above the row header cells and to the left of the column header cells) selects all the cells in the grid.

As an example of a more cell-oriented application, the download code includes an application called SimpleSpread. This application mimics a simple spreadsheet and lets you do summations of the numeric values in a cell. It uses a combination of selection mode and some event handling to

give you a similar selection experience to most spreadsheets—specifically, it acts like a combination of `RowHeaderSelect` and `ColumnHeaderSelect`, even though you can't achieve that through the `SelectionMode` property alone. The SimpleSpread sample application is shown in Figure 6.10.

As you can see, the application lets you enter numbers into the cells; then you can select a sequence of cells and press the *Sum* button in the tool strip control at the top to get it to calculate the sum and place that in the next cell to the right or below the sequence of selections. As Figure 6.10 shows, this application even supports selecting rectangular groups of cells, and it will compute the summation in both the row and column directions. The logic is nowhere near complete to handle all combinations of selections and cell contents, but it gives you a good idea of how to set something like this up.

To code this up (as shown in Listing 6.6), I had to do a few things that are different from your average `DataGridView` application. As I mentioned, I wanted to support a spreadsheet-like selection model, where you can select individual cells, but that selecting a column or row header would select the entire column or row, respectively. To do this, I set the `SelectionMode` for the grid to `RowHeaderSelect`, turned off sorting for all the columns as I created them and added them to the grid, and then handled the `ColumnHeaderMouseClick` event to manually select all the cells in a column when the user clicks on a column header.

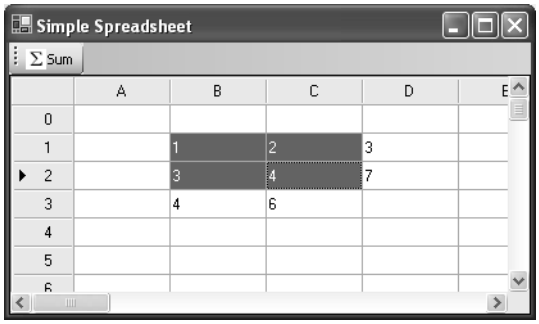

FIGURE 6.10: SimpleSpread Sample Application

LISTING 6.6: Spreadsheet-Oriented Grid Column Selection Support

```
public partial class SimpleSpreadForm : Form
{
   public SimpleSpreadForm()
   {
      InitializeComponent();
      m_Grid.SelectionMode =
         DataGridViewSelectionMode.RowHeaderSelect;
   }

   private void OnFormLoad(object sender, EventArgs e)
   {
      int start = (int)'A';
      for (int i = 0; i < 26; i++)
      {
         string colName = ((char)(i + start)).ToString();
         int index = m_Grid.Columns.Add(colName, colName);
         m_Grid.Columns[i].SortMode =
            DataGridViewColumnSortMode.NotSortable;
         m_Grid.Columns[i].Width = 75;
      }
      for (int i = 0; i < 50; i++)
      {
         m_Grid.Rows.Add();
      }
   }

   private void OnColumnHeaderMouseClick(object sender,
      DataGridViewCellMouseEventArgs e)
   {
      m_Grid.ClearSelection();
      foreach (DataGridViewRow row in m_Grid.Rows)
      {
         row.Cells[e.ColumnIndex].Selected = true;
      }
   }
   ...
}
```

In this case, I just programmatically added some rows and columns to the grid, set the column headers to be the letters of the alphabet, and turned off sorting on the column by setting the `SortMode` property to `NotSortable`. If you were going to support very large spreadsheets, you might need to maintain an in-memory sparse array, and only render the

cells as you need them (which you could do with virtual mode) to avoid the overhead of maintaining a large number of cells, their contents, and their selections if the grid will be sparsely populated.

To get the row numbers to display in the row headers, I handled the `RowAdded` event and set the header cell value in that handler:

```
private void OnRowAdded(object sender, DataGridViewRowsAddedEventArgs e)
{
   m_Grid.Rows[e.RowIndex].HeaderCell.Value = e.RowIndex.ToString();
}
```

Another selection mode you might want to support is to have **hot cells**, meaning that the selection of cells changes as you move the mouse around the grid without having to click. To do this, you could just handle the `CellMouseEnter` and `CellMouseLeave` events, selecting and deselecting the cell under the mouse in those handlers, respectively.

Formatting with Styles

The last topic I want to cover about the `DataGridView` is how to handle custom formatting of cells. As mentioned earlier, the grid supports a rich formatting model. The styles in the grid work in a layered model, which lets you set styles at a more macro level, then refine it at a more micro level. For example, you might set a default cell style that applies to all cells in the grid, but then have one entire column that has a different cell formatting, and have selected cells within that column have yet a different cell formatting. You do this by setting a series of default cell style properties that are exposed on the grid, which you can then refine by setting cell styles at the individual cell level.

As can be seen in Figure 6.11, the lowest layer in the model is the `DefaultCellStyle` property. This style will be used by default for any cells in the grid that haven't had their style set to something else by one of the other style layers. The next layer up contains the `RowHeaders-DefaultCellStyle` and `ColumnHeadersDefaultCellStyle`, which affect the way the header cells are presented. Above that layer sits the `DataGridViewColumn.DefaultCellStyle` property, followed by the

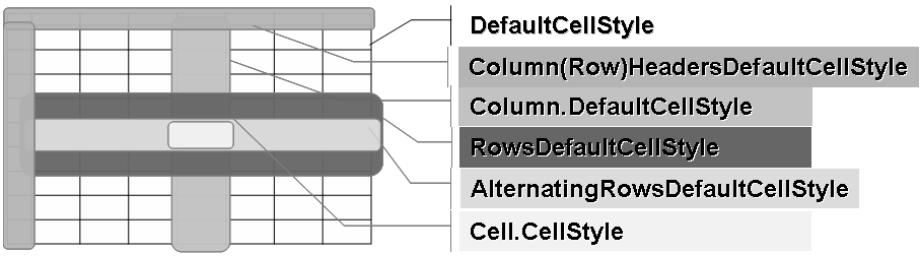

FIGURE 6.11: Cell Style Layering

`DataGridViewRow.DefaultCellStyle` property, representing the default styles on a column-by-column or row-by-row basis. The grid also supports an alternating row cell style that is set through the `AlternatingRows-DefaultCellStyle` property on the grid. Finally, the top-level layer that will override the settings of any of the lower layers if set is the `DataGrid-ViewCell.CellStyle` property.

You can set these properties programmatically by accessing the appropriate property member on the instance of the grid, column, row, or cell. All of these properties are of type `DataGridViewCellStyle`, which exposes properties for setting fonts, colors, alignment, padding, and formatting of values. You can also configure the cell styles through the designer. Whenever you access one of the cell style properties on the grid or a column through the Properties window or Smart Tag property editors in the designer, you will see the CellStyle Builder dialog shown in Figure 6.12.

Using the property fields in this dialog, you can set fine-grained options for how the cell will display its content, and you can even see what it is going to look like in the Preview pane at the bottom of the dialog.

You can also set border styles for cells using the grid's `CellBorder-Style`, `ColumnHeadersBorderStyle`, and `RowHeadersBorderStyle` properties. Using these styles, you can achieve some fairly sophisticated grid appearances, as seen in Figure 6.13. In this sample, default cell styles were set at the column and row level, and then the filling in of the shape was done through individual cell selection.

However, you will still hit some limitations in using cell styles. For example, a natural next step for the grid shown in Figure 6.13 would be to

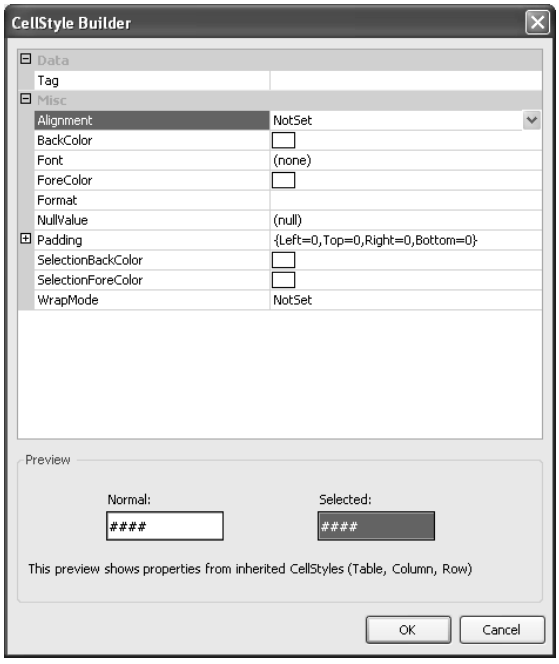

FIGURE 6.12: CellStyle Builder Dialog

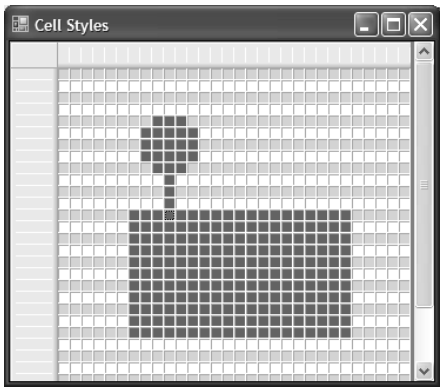

FIGURE 6.13: Cell and Border Styles Applied

set the border colors on the cells that have been colored in to show a black border. However, there is really no way to accomplish this just through cell styles, since the border styles available are only 3D effects and are applied at the grid level for entire cells, not for individual sides of a cell. But, as always, you can almost always accomplish what you need through custom painting or custom cell type definition.

Where Are We?

This chapter has covered all the main features of the `DataGridView` control. It focused a lot on the code involved and what the capabilities were. For most common cases, you'll be able to get what you need simply by defining the columns in the designer, setting up some data binding with the Data Sources window, and maybe mixing in some custom code and event handlers. The `DataGridView` control is considerably simpler to use and more flexible and powerful than the `DataGrid` control from .NET 1.0, and you should always favor using the `DataGridView` for new applications that will present tabular data or something that needs to be presented in a grid format.

Some key takeaways from this chapter are:

- Setting the `DataSource` and `DataMember` properties on the `DataGridView` to a `BindingSource` that binds to the data source is the standard way of using the grid.
- Use the Edit Columns functionality in the designer Smart Tag for the easiest way to edit and customize the bound and unbound column types and properties.
- Bound columns will use automatic type conversion and formatting similar to the `Binding` object, as discussed in Chapter 4.
- To add a custom cell type, you create a custom column type derived from `DataGridViewColumn`, and a custom cell type derived from `DataGridViewCell` or one of the derived built-in cell types.

Next we'll go deeper into the data-binding mechanisms used in Windows Forms, specifically the interfaces that you need to understand and implement in order to create any custom data objects or collections that you want to use for data binding. You will gain a better understanding of what the data-bound controls are looking for when you set up data binding, and how they use what they find.