CHAPTER 2

Building Visual Basic .NET Windows Applications

Windows Application is really inappropriate as a project title, considering that it has the ability to be compiled to IL and ported to another operating system where it can be natively compiled. We'll use this term only because it's the name of the project type in .NET.

Rather than attempting to demonstrate every feature related to developing Windows applications in Visual Basic .NET, we will use key features and tools that will help you see the object-oriented nature of .NET as you work with the standard design components in the Visual Studio .NET IDE. The main focus of this chapter is to introduce you to some key concepts related to .NET development using forms and controls, while looking at their underlying implementation in .NET. In Chapter 5, "Advanced .NET Windows Applications," we will build applications with a wider range of features.

Working with .NET Forms and Controls

The basic unit of development in a .NET Windows application is (still) a form. Each form represents a single window in a running application. We add controls from the Toolbox to enhance a form's user interface.

The .NET version of a form has many new features. In .NET, all languages create a form based on the System.Windows.Forms.Form class in the common language runtime libraries. When I say "based on," I am actually referring to

an OOP concept known as *implementation inheritance*, which allows one type to establish a relationship to another type for the purpose of inheriting functionality. Look at it this way: Visual Basic has always had forms, but we have never had to design into them the ability to be resized or minimized or to display a blue title bar, and so on. All that functionality was inherited from a generic Form object when we added the form to our project. Even in Visual Basic 6 we could declare a variable of type Form, and a new object would be created in memory that had its own Name, hwnd, caption, and so on. Even the first piece of code you see in Visual Basic .NET requires that you have some understanding of Inheritance, as you can see in the Visual Basic .NET code that follows. This code shows the relationship between a form and its immediate base class:

```
Public Class Form1
    Inherits System.Windows.Forms.Form
End Class
```

Forms Are Objects

Visual Basic has always created forms as objects, but until Visual Basic .NET we had not seen any underlying code to prove that. Quite possibly the biggest change in Visual Basic .NET is that all the underlying details of how things are done are no longer abstracted away from us. Although classes are formally discussed in Chapter 6, "Building Class Libraries in .NET," for now you need to understand two basic points. The first point is that all things in .NET are objects. The second point is that all objects are based on a template that a programmer built using code. The most common template for an object is called a class. The description for an *object* in this chapter will be limited to a self-contained entity that has properties to describe it, methods that provide behaviors, and events that allow it to respond to messages.

Form1 is actually a class called Form1 and at runtime, any number of actual Form1 objects can be created based on that class. This chapter focuses on many aspects of designing Windows applications, but as we're using different forms, controls, and other objects, I need you to remember that all these things are built using these object templates.

A Change of Events

Events in Visual Basic .NET are very different from what they were in Visual Basic 6. Event names, the way they are used, and their sheer number have understandably changed in .NET because all controls in .NET were built from the ground up.

For example, the event for a Visual Basic 6 text box that occurred as the user typed into that text box is called the Change event. In .NET, that event is now implemented as the TextChanged event. Besides the name change, the next major difference is that this event now takes sender and e arguments. In Visual Basic we used control arrays when multiple controls needed to share code. In Visual Basic .NET we map two events to the same event handler. Finally, the keyword Handles is used to indicate that a subroutine is to handle the TextChanged event for the text box txtFirstName. Although we might think from our prior experience with Visual Basic that this isn't necessary, keep in mind that the things we're using to build our applications in .NET are based on the CLS and a solid object-oriented foundation. The more you learn about the underlying foundation of .NET, the easier it will be to understand why some things are so different from Visual Basic 6. The following examples contrast the difference between the Change and TextChanged events.

Here is an example in Visual Basic 6:

```
Private Sub Text1_Change()
End Sub
```

Here is an example in Visual Basic .NET:

```
Private Sub txtFirstName_TextChanged(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles txtFirstName.TextChanged
End Sub
```

Classroom Q & A

Q: If the .NET version of this code says that it handles the TextChanged event for txtFirstName, could I name this subroutine anything I want?

A: In Visual Basic we just write the code in the event and don't differentiate between the event and event handler. Here, we have a predefined set of events that can be raised in an object and we need to write custom handlers to handle those events. We can name it whatever we want.

Q: So, I take it that sender is kind of like the index parameter from Visual Basic?

A: Well, in a way. It is used to identify which object received the event that caused the event handler to execute. By accessing properties of the sender object, we can find out exactly which control received the event.

Q: How do we make it so that more than one control can use the same event handler?

A: You just append more information to the Handles part of the event handler declaration. Lab 2.1 demonstrates that process.

Lab 2.1: Working with Event Handlers

This lab demonstrates the association between events and event handlers. We will create three text boxes. The first two contain the first and last names of an individual. As either the first or last name is changed, the third text box will dynamically display the resulting email address. To start, we will use separate TextChanged events for capturing changes to the first and last names but will quickly change our implementation to a more efficient event handler.

Perform the following steps:

1. Start a new Visual Basic Windows application.
2. Drag three TextBox controls and three Labels to the form.
3. Change properties for controls by selecting them at design time and entering a new value for the desired property in the Properties window. Change the following properties for the six controls you just added to Form1.P: txtFirstName_TextChanged to txtLastName_TextChanged and test the application again. This time the results were correct, but the method we used to obtain those results used redundant code.
4. Remove the entire txtLastName_TextChanged event handler.
5. Change the name of the txtFirstName_TextChanged event handler to Names_TextChanged.
6. Add the following code to the end of the declaration for Names_TextChanged so that the same event handler handles the TextChanged events for both text boxes.

```
, txtLastName.TextChanged
```

What you should see is:

```
Private Sub Names_TextChanged(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles txtFirstName.TextChanged, _
txtLastName.TextChanged
    txtEmail.Text = txtFirstName.Text & "." & txtLastName.Text &
_
"@somecompany.com"
End Sub
```

7. Run the application again. Functionally, it still works the same but with half the code. This event-handling process uses a concept known as delegation, discussed in Chapter 7, "Extending Visual Basic .NET Classes."

For now, let's just view our Names_TextChanged event handler as a method of the Form1 class that handles the TextChanged event for two text boxes. The next step will be to get used to the different controls and events used in building Windows forms.

The Controls Collection

Each form has a collection called Controls, which contains all the controls on that form. The Contains method (not specific to a form) returns a boolean value that indicates whether one control is contained in another.

```
Dim i As Integer
For i = 0 To Me.Controls.Count - 1
If Me.Controls(i).Contains(mycontrol) Then
   MessageBox.Show(Me.Controls(i).Name)
End If
Next I
```

Most of the standard controls used on Windows forms are part of the System.Windows.Forms assembly in System.Windows.Forms.dll.

Familiar Controls or Just Familiar Looking?

Many controls that look identical to their predecessors act quite differently in .NET. Let's take the scrollbar controls as an example. The first thing you'll see regarding scrollbars in .NET is that instead of having a single control with an orientation property, the .NET Toolbar has separate vertical (VScrollBar) and horizontal scrollbar (HScrollBar) controls. The scrollbars have Minimum and Maximum properties that are typically mapped to the target range they are used to scroll through. These properties used to be Min and Max in Visual Studio 6. The following exercise demonstrates much more than scrollbars. In Lab 2.2 you're going to use HScrollBar controls to see how even the simplest of tasks have changed for Visual Basic programmers.

Lab 2.2: Controls in .NET

.NET controls are smarter than their predecessors. They can adapt to their surroundings. We'll be using three HScrollBar controls to control the background color of the current form by individually setting the red, green, and blue values (RGB) of a Color object.

Perform the following steps:

1. Start a new Visual Basic Windows application.
2. Drag a GroupBox (used to be called a Frame control) to Form1.
3. Drag three HScrollBar controls inside the GroupBox.
4. Because each of these HScrollBar controls will be setting the RGB values, we need to set their Minimum and Maximum values to match the possible values for RGB. For each control, set the Maximum property value to 255 and leave the Minimum property value at 0. The HScrollBar1 control will change the red, HScrollBar2 will change the green, and HScrollBar3 will change the blue value of a Color object. Because RGB values are always in red, green, blue order and have values ranging from 0 to 255, a value of 0,0,255 indicates pure blue.
5. Add the following event handler to handle the Scroll event for all three HScrollBar controls:

```
Private Sub ScrollAll(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.ScrollEventArgs) Handles HScrollBar1.Scroll,
HScrollBar2.Scroll, HScrollBar3.Scroll
Me.BackColor = Color.FromArgb(HScrollBar1.Value,
HScrollBar2.Value, HScrollBar3.Value)
End Sub
```

Color is a class defined in System.Drawing. If you look at the References section in the Solution Explorer, you will see that our application has a reference to the System.Drawing assembly. That is why we can use the Color class.

6. Run the application. Use the scrollbars to change the BackColor of Form1, but watch the GroupBox controls color! The GroupBox has the inherent ability to change the color of its border in relation to its immediate container, as shown in Figure 2.1. We deduce from this that there is an event that notifies the GroupBox that such a change was made. The System.Windows.Forms.Form and System.Windows.Forms.Control classes have many events we're not used to seeing

yet, such as ForeColorChanged and BackColorChanged that make features like this possible. All of our controls and forms are based on (inherited from) these two classes.

If you're starting to believe that I'm trying to get you used to the object-oriented nature of .NET, you're right! If the objects, classes, and inheritance are all still confusing, don't be discouraged. We haven't even formally explored those concepts yet. By the time we get to a discussion on classes and objects, you will have quite a few examples to fall back on. For now, understand that we are in a pure object-oriented environment and nothing is abstracted away from us as it was in Visual Basic 6. These two truths will drive your application development in .NET more than you may realize at this point, but be excited; with this environment comes a developmental power and elegance previously lacking in Visual Basic.

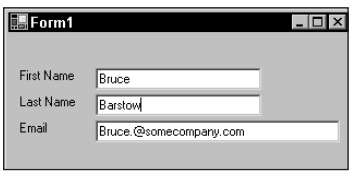

Figure 2.1 The GroupBox control dynamically adjusts to its parent's BackColor property by changing the color of its own border.

Classroom Q & A

Q: I have used quite a lot of objects in the past, but I'm a little confused by the syntax used with the Color object. I don't see how we have a copy, *instance* I think you called it, of the Color object and yet we are calling a method on it.

A: First of all, FromARGB() is a static method in the base class Color. You should remember that Form1 inherits many methods and properties from the Form class and can redefine what those methods (like Load) do for each specific form we create. That makes Form the *base class* for Form1. Well, Color is the base class for a new Color object, but the Color base class has functionality that doesn't make much sense to be overwritten in each of the colors created based on the Color class. FromARGB() is an example of such a method and is referred to as a static method. *Static* methods are not called as object.methodname, but rather, as baseclass .methodname.

Q: Is this explained further in this book?

A: Yes, I talk about this more in Chapter 6. We just haven't had to learn a lot of these concepts in Visual Basic because it was focused on being a very friendly, productive environment with a lot of the detail abstracted away. Now we have the friendly environment with all the power of an object-oriented language. Think of it as changing your tools in your Toolbox. We just got rid of our Toolbox and replaced it with a roomful of the latest power tools. By showing you what's new while I'm using a concept like scrollbars, text boxes, and labels that have hardly changed at all, we can use what *is* familiar to us as an anchor for our OOP learning process.

Locations and Size in .NET

In Visual Basic 6, we used the Move method or the Left, Top, Width, and Height properties to move things to the desired location. In .NET we do things (of course) a little differently.

The older Left and Top properties *seem* to have been replaced with the more universally correct representation of an x- and y-axis value pair known in .NET as the Location property. The Left and Top properties still exist, but they are not in the Properties window. To move a control to the upper-left corner of a form we could set the control's Location property using the values of 0 and 0.

```
Button1.Location = New Point(0, 0)
```

Here, I've created a new Point object and initialized it by setting both its x- and y-axis to zero. If you look at the IntelliSense in Figure 2.2, you will see that the Location property is very clear about needing a Point object instead of just two numbers.

Setting Width and Height

Setting the Width and Height properties also offers a small surprise. If you set these two Integer values thinking about the twip unit of measurement from Visual Basic, you'll have some huge controls. The default unit of measurement is expressed in pixels in .NET.

```
Button1.Width = 100  'pixels
Button1.Height = 40  'pixels
```

Double their size:

```
Button1.Width *= 2   'equivalent to Button1.Width = Button1.Width * 2
Button1.Height *= 2  'using C++-style shorthand
```

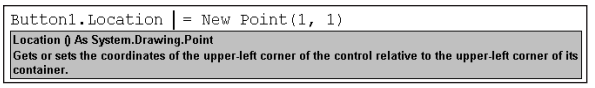

Figure 2.2 Pay attention to what IntelliSense is telling you in .NET. The IntelliSense here indicates that the Location property requires a Point object.

Setting the Form Size Using the Size Property

Setting the Size property requires the assignment of a Size object. Restated, that means that the Size property is a property of type Size. The following code creates a new Size object to be used in the assignment:

```
Me.Size = New Size(100, 100)
```

Strangely enough, when the form is maximized, this code will apparently do nothing, but when the form is back in Normal mode, it will have a width of 100 and a height of 100.

Controlling How Large or Small a Form Can Be

The Form object uses the MaximumSize and MinimumSize properties to limit how small or large a form can be. These values have no effect on the ability to maximize or minimize the form using the minimize and maximize buttons. These values control how small or large the form can be by using the handles to expand and contract the form's borders.

The SystemInformation object can be used to determine not only the minimum window size, but also things such as the user name, the domain the user is logged into, whether the user has a network card, how many buttons the user's mouse has, and even the system boot mode.

Docking and Anchoring

Controls have a Dock property that enables them to be permanently fixed to different sides of a container. The Anchor property for a control is used to ensure that the anchored edges maintain a position relative to the edges of the container. Figure 2.3 shows the result of setting the Dock property for Button1 to Top.

Affecting Visibility

Forms can be manipulated with the standard Visible property and the Hide and Show methods from Visual Basic 6, but there are some new features to note.

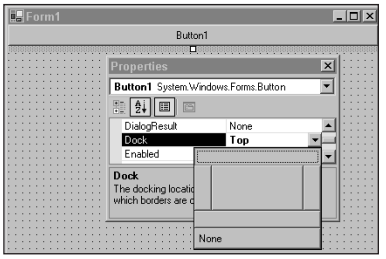

Figure 2.3 Controls are anchored using the Dock property.

Keeping a Form on Top

The TopMost property of a form can be used to ensure than no matter what is displayed, the TopMost form will always be on top of any other form. The following code creates a new form beneath the current form:

```
Me.TopMost = True
Dim myform As New Form()
myform.Show()
```

Changing a Form's Opacity

A form can be made partially visible (or partially invisible for you pessimists) by using the Opacity property. This property normally has a value of 1 to indicate that it appears normal (that is, opaque not transparent). The Opacity value of 0 indicates that the form is completely transparent. The following code causes the current form to fade away and then return. Figure 2.4 shows a form at 50 percent (.5) transparency.

```
Dim i As Double
For i = 1 To 0 Step -0.01
     Me.Opacity = i
Next
Me.Opacity = 1
```

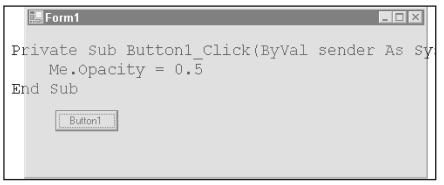

Figure 2.4 A form with an Opacity setting of .5. is exactly half visible.

The standard For loop used here iterates through its code between the possible values for opacity from 1 to 0, decrementing by –0.1 until the current form is completely transparent. For loops have an incremental Step value of 1 when not specified.

.NET Controls

This section is not intended to show you every control in .NET. Actually, my only intention is to get you familiar with the environment and how different things really are beneath the surface of controls that seem like old friends. Now that we have some of the basic principles in our skill set, we can start working with a few traditional controls, look at what's new in them, and progress to some of the controls that are unique to .NET. All the following code applies to the ListBox, ComboBox, and other list-related controls in .NET.

The ListBox Control

Every Windows programmer has used a ListBox control before, but the changes to the ListBox in .NET have a lot to do with what is underneath the control. Many objects have collections of things such as images, buttons, controls, or the Items collection in a ListBox. In .NET, collections aren't created differently for each object but rather inherited from a common collection interface.

The Items Collection

Any object that has a collection in .NET will most likely inherit its methods from ICollection and IList. These two things are known as interfaces in the common language runtime libraries. By having all collections inherit a defined set of methods from these interfaces, we can expect any collection in .NET to have an expected interface.

The interface for a car usually doesn't vary from car to car for its basic, expected functionality, although each car can have extended features. We expect to enter all cars via a door, we expect to start all cars with an ignition key, we press an accelerator to make the car go, and we press a brake pedal to make the car stop. The point is that by having an *expected interface* for every example of a given type, the user of each item (type) can use it intuitively. Doing something unexpected in the design, such as putting the brake pedal on the right, could mean that the user of the vehicle might never want to attempt using that model again.

In .NET collections, this translates to every collection having an Add method, a Remove method, a Count property, and an Item Property that we instinctively look for as programmers using those objects. As a matter of fact, after a very short while as a .NET programmer, you will quickly become agitated by programmers who reinvent the wheel instead of supporting a standard interface.

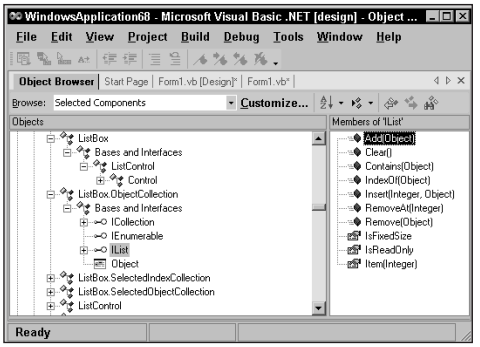

Figure 2.5 ListBox.Items is actually based on the class ListBox.ObjectCollection that implements the ICollection and IList interfaces.

Looking at the Object Browser in Visual Basic .NET, we can see on what classes and interfaces each of our objects are based. Figure 2.5 shows the classes and interfaces implemented by a ListBox. The right panel in Figure 2.5 shows the methods and properties of the IList interface.

The Items collection represents the items stored in the ListBox. To add values to any collection that is based on these well-known interfaces, we use the Add method as shown in the following code:

```
ListBox1.Items.Add("Azure")
ListBox1.Items.Add("Crimson")
ListBox1.Items.Add("DarkOrchid")
ListBox1.Items.Add("DodgerBlue") 'yes that is actually a built-in color
```

Common Collection Methods and Properties

All Collections have at least the properties and methods listed in Table 2.1.

Changing the Selected Item in a ListBox

In earlier forms of Visual Basic we would use the Click event to write code to capture the newly selected item. In .NET we write an event handler for the SelectedIndexChanged event of the ListBox. The SelectedIndex property can be used to pinpoint which item was selected. The following code shows an example of a SelectedIndexChanged event handler:

```
Private Sub ListBox1_SelectedIndexChanged(ByVal sender
As System.Object, ByVal e As System.EventArgs) Handles
ListBox1.SelectedIndexChanged
     Me.ListBox1.ForeColor =
     Color.FromName(ListBox1.Items(ListBox1.SelectedIndex))
End Sub
```

Table 2.1 Collection Properties and Methods

PROPERTY OR METHOD	DESCRIPTION
The Count property	Represents the number of items in the collection
The Add method	Add items to a collection
The Remove method	Remove items from a list
The Item property	Used to refer to a specific item in a collection

When an item is selected, the text value in the ListBox is used as the argument for the FromName method on the Color class to set the ForeColor on the ListBox as shown in Figure 2.6.

Taking Advantage of the Collection Interfaces

Although showing you the underlying interfaces and classes of .NET may seem premature, knowledge of the underlying common language runtime class libraries will be the single most defining factor in your mastery of .NET. We can already take advantage of the knowledge we have by looking at the AddRange method of a ListBox. The AddRange method for a ListBox takes an ObjectCollection as its only argument. Because we know that the Items Collection is an example of an ObjectCollection, we should be able to add the contents of one ListBox to another by passing the Items collection of one ListBox to the AddRange method of another ListBox as shown here using Visual Basic .NET:

```
ListBox1.Items.Add("Test Data1")
ListBox1.Items.Add("Test Data2")
ListBox2.Items.AddRange(ListBox1.Items)
MessageBox.Show(ListBox2.Items(0).ToString())
```

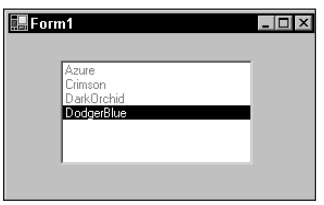

Figure 2.6 Clicking on a ListBox item causes the SelectedIndexChanged event.

Classroom Q & A

Q: How many classes and interfaces are there in .NET?
A: I have no idea, but there are many. We end up using more and more as we build broader applications. We are going to use more than you will remember after reading these chapters, but you will have learned a certain set of behaviors and patterns related to the libraries that will make using them much easier and intuitive. Don't worry if it seems intimidating now; we haven't even officially introduced the concept of an interface yet.

Q: In your last example, why did you need to use ToString in the last line of code?
A: You need to be sure what you are sending to a method is the right type. MessageBox.Show needs a String, so make sure you treat the value at Items(index) as a String using the ToString method.

Q: When I tried it without the ToString method, it still worked.
A: It's good to experiment and you're right; it does work without the ToString() method, but Visual Basic doesn't always do things for you. My first two golden rules of programming are always be explicit and never let a compiler do you a favor. It is considered a better programming practice to indicate explicitly how values are to be treated in an expression.

The CheckedListBox Control

The CheckedListBox control is new to .NET. Each item in the control appears with a check box preceding it. Much of the code we use for other list-related controls still applies. but we'll need to look at a few new methods, events, and properties to make use of this new control.

Adding Items to a CheckedListBox

Because the Items collection for this control also indirectly implements the IList interface, it has an Add method for adding items. But implementing an Add method is done differently for different objects. This particular Add method has three versions we could use. Different versions of a member with the same name are referred to as *overloaded methods*, which we will discuss later. Figure 2.7 shows how the most common version of the Add method, which takes an object followed by a boolean value, controls whether the new item's check box should be selected. That shouldn't be too hard to code because object is actually a reference to System.Object, which applies to every type in .NET and a boolean value is just True or False.

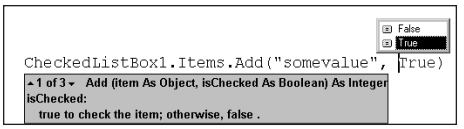

Figure 2.7 Version 1 of 3 for the CheckListBox.Add() method.

note **In Visual Basic 6, any nonzero value was True. That is not the case in .NET. Only True is True and only False is False.**

The following code shows how items can be added to a CheckedListBox control named clb1 with the first three values checked and the last value unchecked:

```
clb1.Items.Add("Architect", True)
clb1.Items.Add("Consultant", True)
clb1.Items.Add("Trainer", True)
clb1.Items.Add("PogoStick Champion", False)
```

As with all collections in .NET, the first position (or index) is zero and the collection has *Count* number of items.

Determining Which Items Are Checked

The CheckedItems property procedure returns an object of type CheckedItem-Collection. We can iterate through unfamiliar collections in a very generic way using an iterator of type Object (see Figure 2.8). It is always more efficient to use an iterator of an appropriate type for the given collection, but an item is actually a property that returns a generic object.

```
Private Sub btnGetCheckedItems_Click(ByVal sender As System.Object,
ByVal e As System.EventArgs) Handles btnGetCheckedItems.Click
   Dim o As Object
   Dim msg As String
   For Each o In clb1.CheckedItems
      msg += o.ToString & Convert.ToChar(13) 'chr(13), vbcrlf still work
   Next o
   MessageBox.Show(msg, "By Item:")
End Sub

Private Sub btnGetCheckedIndices_Click(ByVal sender As System.Object,
ByVal e As System.EventArgs) Handles btnGetCheckedIndices.Click
   Dim o As Object
   Dim msg As String
   For Each o In clb1.CheckedIndices
      msg += o.ToString & Convert.ToChar(13)
   Next o
   MessageBox.Show(msg, "By Index:")
End Sub
```

Figure 2.8 Items in a CheckedListBox can be interrogated for their checked status by Index or by Item.

Changing the Click/Checked Behavior

I was pleasantly surprised to find that I could cause the items to be checked by a single click instead of the default. The following Visual Basic .NET code causes an item to be checked on a single click:

```
checkedListBox1.CheckOnClick = True
```

Sorting

Even though a property seems to be similar to a property we might have become used to in Visual Basic 6, take nothing for granted. Some things are more flexible and some are more restricted. The Sorted property of the ListBox type of controls in Visual Basic 6 was read only at runtime, but the same property can be set at design or runtime in .NET as shown here:

```
checkedListBox1.Sorted = True
```

Other Useful Properties and Methods

Many properties for the ListBox line of controls simply didn't exist as features before .NET. The properties and methods in Table 2.2 are particularly useful.

Table 2.2 ListBox Controls in .NET

PROPERTY OR METHOD	DESCRIPTION
The ClearSelected method	Clears selection without affecting checked status.
The Contains method	Determines if a control is in this container.

Table 2.2 *(continued)*

PROPERTY OR METHOD	DESCRIPTION
The FindString method	Searches for partial match to Item data; second overload for this method allows searching to start at a specific index. FindString("Train") would return the index of "Trainer" in our example.
The FindStringExact method	Searches for exact match of item data. FindStringExact("trainer") would return the index of "Trainer," but FindStringExact("Train") would return a value of –1.
The SelectedIndex property	Returns the index of the selected item.

Dialog Controls

In Visual Studio 6 we typically used a single Common Dialog control to handle all our dialog needs. This created a control that had far too many methods and properties that weren't related to the task we were performing at the time. In .NET we have separate controls for each of the standard dialogs. Table 2.3 shows examples of the separate dialog controls in Visual Studio .NET.

Using the Dialog Controls

The ColorDialog control is activated for display by using the ShowDialog() method. The ColorDialog window is shown in Figure 2.9.

Table 2.3 Dialog Controls in Visual Studio .NET

DIALOG	ALLOWS THE USER TO
ColorDialog	Pick a color from a color palette
FontDialog	Choose a font and apply formatting
OpenFileDialog	Locate a file to be opened
PageSetupDialog	Display page formatting dialog box
PrintDialog	Control and start printing
PrintPreviewDialog	Preview print jobs
SaveFileDialog	Indicate a file and path for saving

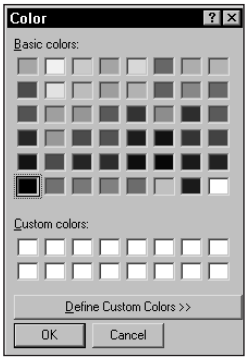

Figure 2.9 The ColorDialog control's Color property is set when the dialog box closes.

After the user selects a color and closes the ColorDialog window, the control's Color property holds a valid Color object. Printing this Color object with ToString()reveals that some colors have proper names such as Chartreuse, whereas others have ARGB values:

```
Color[A=255, R=255, G=0, B=128].
```

The Color property can be used as expected:

```
ColorDialog1.ShowDialog()
Me.BackColor = ColorDialog1.Color
```

Each dialog control sets different values when they are closed. For the SaveFileDialog control, the FileName property is set to the file indicated by the user inside the Save dialog window.

The DataGrid Control

The DataGrid control allows us to view and modify records from a database. Chapter 5, "Advanced .NET Windows Applications," expands on data access in .NET; here we will focus on the DataGrid control and some controls that support its use. Lab 2.3 demonstrates the use of this control.

Lab 2.3: Using Data Access Controls in .NET

This lab demonstrates the use of the DataGrid control and several underlying objects associated to data access. This lab requires a database from the .NET QuickStart samples to be installed. The objective of this lab is to introduce you to the DataGrid control and expose you to the concepts of the Data Adapter, DataSet, and Connection objects.

Perform the following steps:

1. Start a new Visual Basic Windows application.
2. Add a DataGrid control from your Toolbox to Form1. This control is used to display records from a database.
3. On the Toolbox, you should see several tabs: Windows Forms, Components, General, and Data. Select the Data tab on your Toolbox.

   The Data tab contains objects that represent objects in the System.Data and System.Data.SqlClient assemblies. These controls make data access easy by abstracting the details of data access.

note **If you cannot see the tabs I described in step 3, change your current view to Designer view by selecting Designer from the View menu.**

4. Add an OleDbDataAdapter control from the Data tab of the Toolbox to Form1. The control appears in the Control Tray underneath the current form. Unlike Visual Basic 6, controls that are not visible at runtime don't reside on the form surface at design time in .NET.
5. In the Welcome to the Data Adapter Configuration Wizard screen, click Next.
6. When confronted with the Choose Your Data Connection screen, click the New Connection button to display the Data Link Properties window. We are using the Microsoft Jet provider to indicate that we are working with a Microsoft Access database (see Figure 2.10).

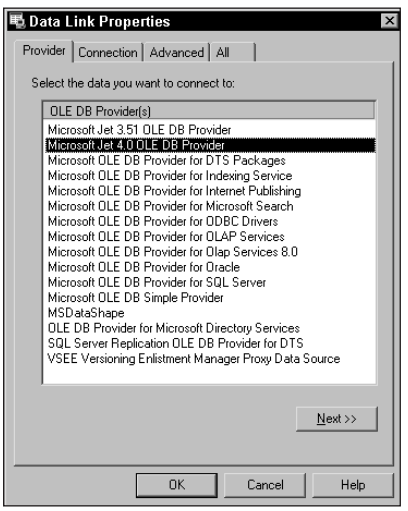

Figure 2.10 Select the Microsoft Jet provider when working with Microsoft Access databases.

7. From the Provider tab of the Data Link Properties window, select Microsoft Jet 4.0 OLEDB Provider.
8. Click Next to move to the Connection tab or just select the Connection tab on the Data Link Properties window.
9. Assuming you have installed the Quickstart samples, use the button to the right of the Select or Enter a Database Name text box to browse for and select the grocertogo.mdb database. This database will be buried quite a few levels deep, so use the search tool in windows to find the database.
10. Click OK to return to the Choose Your Data Connection window.
11. Click Next to advance to the Choose a Query Type window.
12. Click Next to accept the default of Use SQL Statements and to advance to the Generate the SQL Statements window.
13. Type Select * From Products into the only text box on the Generate the SQL Statements window.

The Query Builder button can be used to experiment with the Structured Query Language (SQL) and to build more complex queries. SQL has become the standard language for communicating with relational databases.

14. Click Finish to end this Wizard. A new object should have appeared in your control tray beneath Form1. OleDbConnection1 is an object that represents a connection to a database.
15. Select the OleDbConnection1 object and view its ConnectionString property using the Properties window. This property has most of the information needed to locate and connect to your database.
16. Be sure you are in Designer view (as opposed to code view). Use the View menu to change between the two views if necessary. Switching to Designer view causes the Data menu to appear in Visual Studio .NET.
17. Choose Generate DataSet from the Data menu.
18. Nothing needs to be changed on the Generate DataSet window, so click OK. This creates yet a third type of control in the control tray that represents a DataSet.

The underlying objects we're using are as follows:

Connection object. Represents a database connection.

Data Adapter object. Uses a connection and a command to fill a DataSet with Records.

DataSet. Represents a set of records that were requested using a command. Our command was Select * from Products, which asks for all records (*) from the Products table.

To associate the set of records returned from our SQL query with the DataGrid control we have to set the DataSource property. Establishing an association between a control and a database is referred to as *binding* a control to data.

Continue on and bind the DataGrid control to the DataSet.

19. Select the DataGrid control and set its DataSource property to DataSet11.Products as shown in Figure 2.11.

20. Double-click Form1 to open the Code Editor window, focused on the Form1_Load event handler. Enter the following code for form's Load:

```
OleDbDataAdapter1.Fill(Me.DataSet11)
```

21. Run the application, and your DataGrid control should display all the records from the Products table (see Figure 2.12).

Navigation, modification, addition, and deletion of records are possible with this control. You'll be sadly disappointed if you attempt to code the Click, Navigate, or KeyDown event to respond to the user clicking on different fields or using the arrow keys to move from one record to the next. Theses events don't do a very good job of detecting user interaction. Chapter 5, "Advanced .NET Windows Applications," introduces you to the underlying concepts surrounding data access in .NET.

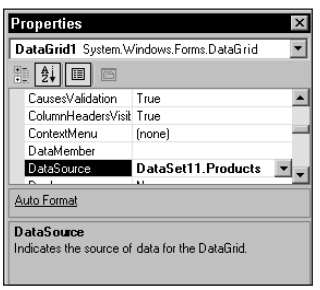

Figure 2.11 Use the arrow next to the DataSource property to select from valid data sources on the current form.

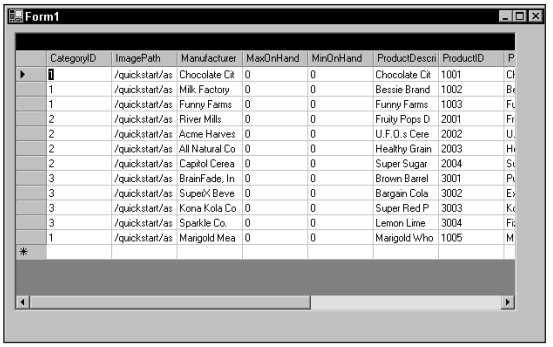

Figure 2.12 The DataGrid control displays records based on its DataSource.

The Panel Control

The Panel controls are dockable containers in .NET. They work like frames in HTML development. These panels are used as viewing panels (or panes if you like) to separate sections of our user interface. Figure 2.13 shows a form with a single panel whose BorderStyle property has been set to Fixed3D and whose Dock property has been set to Left. Panels are invisible, by default, at runtime.

The DateTimePicker and MonthlyCalendar Controls

The DateTimePicker is possibly the simplest of the controls. To use it, just place it on a form. Users are presented with a drop-down calendar from which they can choose a date. The MinDate and MaxDate values can be used to define a valid range of dates. After a date has been chosen, the Value property of the control will contain that date. The Format property can be set to Long, Short, Time, or Custom to change the format in which the date or time is stored and displayed. The MonthlyCalendar control displays a calendar page for the current month with the current day circled and special dates in boldface type. The DateTimePicker control is used to select a date, whereas the MonthlyCalendar control is used to display the date via a calendar page.

Figure 2.13 Panel controls allow you to separate or even hide sections of a user interface.

Dynamically Adding Controls

Adding controls at runtime (dynamically) is a little different than it was in Visual Basic 6. The following code creates a control, but it never appears:

```
Dim mybtn As New Button()
mybtn.Location = New Point(10, 10)
mybtn.Show()  'DOES NOTHING
mybtn.Visible = True 'DOES NOTHING
Me.Controls.Add(mybtn) 'This is what we want!
```

Controls must be associated with a container and added to the container's control collection. To add a control dynamically to a panel named Panel1 on Form1, use the following code:

```
Me.Panel1.Controls.Add(mybtn) 'bingo!
```

That method works for all controls that have a Controls collection. In the first example, the attempts to use the Show method and the Visible property do nothing and should be omitted.

Scrollable Forms

If we keep running the code in the previous example and add a little code to make each successive button appear underneath its predecessor, eventually we will have controls that cannot be displayed on the form. That was always a point of contention many people have had with Visual Basic forms. In .NET, however, we have only to set the AutoScroll property for a form to resolve that issue. I wanted to see just how far the form would scroll if I kept creating controls.

First I changed the AutoScroll property for the form to True. Then I created a private variable inside Form1 to keep track of the next control's y-value:

```
Private y As Integer = 10
```

Next, I added a button with the following code:

```
Private Sub Button1_Click_1(ByVal sender As System.Object,
ByVal e As System.EventArgs) Handles Button1.Click
   Dim mybtn As New Button()            'create a new button object
   mybtn.Location = New Point(10, y)    'create 10 pixels down from last
   mybtn.Text = Me.Controls.Count - 1   'number the buttons
   Me.Controls.Add(mybtn)               'display the new button
   y += 30                              'set next button's y value
End Sub
```

note **Somewhere near 1,000 buttons, things went haywire, the controls stopped displaying, and the scrollbars didn't work as expected. I would recommend keeping your form within some respectable limitations on its size.**

As shown in Figure 2.14, if even a portion of a control is off the viewable portion of the form, the appropriate scrollbars appear.

Multiple Forms

In applications such as Word, Excel, and even Visual Studio, we have a container environment that holds our documents in Word, our workbooks in Excel, and our design components in Visual Studio. These *container environments* are actually forms themselves. They are a parent form in which all other forms reside. Yes, what you know to be Microsoft Word is simply a parent form that contains children you know as documents. This type of application is referred to as an Multiple Document Interface (MDI) application. What we are using in this chapter are referred to as Single Document Interface (SDI) applications. We will implement MDI applications in Chapter 5.

Adding New Forms

Adding new forms is similar to Visual Basic 6: Just select Add Windows Form from the Project menu, select Windows form, and click Open. The difference is where to go to set the startup form. One minute you'll see Properties appear as the last menu item in the Project menu, and the next minute it disappears. The trick is that you either have to right-click the project in the Solution Explorer or select the project name and choose Properties from the Project menu. Once in the properties page for the project, set the startup object to the appropriate form.

tip **Properties only appear under the Project menu when a project is selected in the Solution Explorer.**

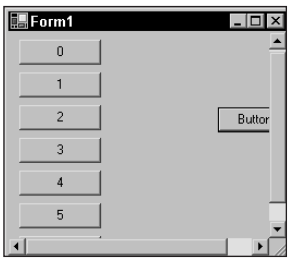

Figure 2.14 AutoScroll adds scrollbars to forms when any control's dimensions are not completely contained in the form's viewable area.

The Anatomy of a Form

By now, I'm sure you've noticed there are things in a form beyond what we've talked about. Most of the information that used to be hidden away in .frm files in Visual Basic 6 appears in our code in Visual Basic .NET. The plus and minus signs allow us to display or hide any section of code. The section, Windows Form Designer generated code, contains all the behind-the-scenes code that controls the creation of our form and the controls on it and is hidden by default.

```
Public Class Form1
    Inherits System.Windows.Forms.Form
       + Windows Form Designer generated code
End Class
```

New()

The first piece of code within that section is a method New. New is a special method for a class called a constructor. In Visual Basic 6, we would think of this as the initialize event for a class. A constructor controls how a class is created.

Dispose()

For those of you who have heard enough about interfaces, Dispose() is used to clean up memory used by the form. For those who want to know more, this method will most likely exist in every class you ever create. Dispose() is part of the IDisposable interface. Objects made from classes that support the IDisposable interface have a much better chance of avoiding delayed recovery or loss of resources.

InitializeComponent()

InitializeComponent represents all the code to define, default, and create your controls on the form. The code here and in the other methods in this section should not be modified directly unless you are very comfortable with what you're doing.

Classroom Q & A

Q: Okay, so you said that Dispose() was a method that was part of the IDisposable interface, but I don't see any reference to this form inheriting that interface.

A: I think proving it will require jumping into the Object Browser, as you might discern from Figure 2.15. You'll notice that the declaration of the Dispose method in Form1 uses the Overrides keyword. That means that the Form class that Form1 inherits from also has a Dispose method, and we are overriding that functionality locally.

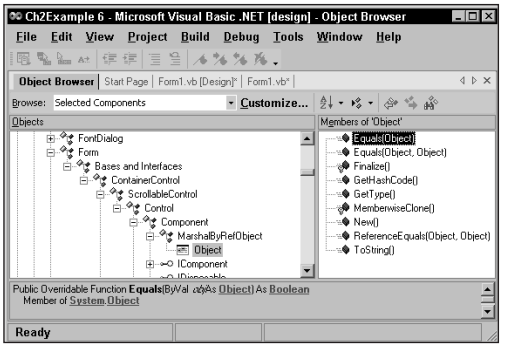

Figure 2.15 The Form class inherits and overrides a Dispose method from a class that is five levels up an object hierarchy.

This is expected in that each object needs to dispose of its own resources in memory accordingly. Figure 2.15 shows how far up the class hierarchy you have to go to see exactly where IDisposable was first inherited.

Summary

This chapter introduced you to more than a few controls and new form tricks. The fundamentals of the relationships between the objects we use in .NET were a strong underlying theme. We will use many more objects and eventually create not only our own controls but also our own libraries of classes. As we progress, use the Object Browser and ILDASM to explore and discover information about the objects you are using.

Review Questions

1. Where does a new form get its default functionality on creation?
2. What keyword is used to map an event handler to a control's event?
3. What happened to the Caption property we used in Visual Basic 6?
4. What is a text box's Change event now called?
5. What method did we use to determine if one control is a parent to another control?
6. Where is the Button class defined?
7. Where is the Color class defined?
8. What property allows you to set an object's *x* and *y* positions?
9. A form's Width and Height can be modified using the Size property. What must be passed to the Size property?
10. How can you set a form to 50-percent transparent?
11. How do you change the startup form in a project?
12. What property can be used to make a Panel visible at runtime?
13. Which of these methods causes a dynamically created control named Button1 to appear on a form?
    a. Me.Container.Add(Button1)
    b. Me.Controls.Add(Button1)
    c. Me.Add(Button1)
    d. Me.Container.Controls.Add(Button1)
14. When will scrollbars appear on a form?
15. What is the purpose of the New() method declared in the Windows Form Designer generated code section of a form?
16. Location is a property of what type?
17. How do you populate a data set with records?
18. What is the property for a control to cause it to be bound to data?
19. Where can you find the OleDbDataAdapter control?
20. What does Select * From Employees return when it is used as the command sent to a relational database?

Answers to Review Questions

1. New forms inherit from the System.Windows.Forms.Form class.
2. The Handles keyword is used to map an event handler to one or more events.
3. Visual Basic .NET uses only the Text property, whereas Visual Basic 6 used Caption for some controls and Text for others.
4. Text boxes in .NET have a TextChanged event to replace the older Change event.
5. The Contains method can be used to determine if a control is a child (contained in) of another control.
6. The Button class is defined in System.Windows.Forms.Button.
7. The Color class is defined in System.Drawing.Color.
8. The Location property allows you to set an object's *x* and *y* positions on the screen.
9. A form's Width and Height can be modified using the Size property by assigning a Size object.
10. A form can be made 50-percent transparent by setting the form's Opacity property to .5.
11. You can change the startup form in a project by right-clicking on the project name in the Solution Explorer, selecting Properties, and changing the startup object to the desired startup form.
12. BorderStyle is one property that can be used to make a Panel visible at runtime.
13. **b.** Me.Controls.Add(button1) will cause Button1 to appear.
14. Scrollbars appear on a form when the form's AutoSize property has been set to True and at least one control is partially outside of the viewable portion of the form.
15. The New() method declared in the Windows Form Designer generated code section of a form is the constructor of the class defining that form. A constructor for a class acts like an initialize event in a sense but is primarily responsible for the actual creation of objects based on that class.
16. Location is a property of type System.Drawing.Point.
17. A DataSet is populated with records by using the Fill method on a data adapter. Although different data adapters can be used, we used the OleDbDataAdapter.
18. The DataSource property is the property we used to cause a DataGrid control to be bound to data. Other properties might be required for different controls we use later.
19. The OleDbDataAdapter control is located on the Data tab of the Toolbox.
20. Select * From Employees returns all fields of all records in the Employees table.