Chapter 4
Basic Controls

In this chapter we will explore the use of some popular and commonly used controls. Controls can be placed directly on your form or in a dialog. A dialog is just a form configured to more closely represent the standard dialog of Windows applications. There are many controls that you can use and controls have many advanced properties. I want to concentrate on the most commonly used subset of these controls and just their basic properties. This set of controls represents the vast majority of controls you will find used in most applications. This will get you going on real programming applications. By the way, if you can’t find a control you like in Visual Studio 2005 you can look on the web for third party controls. Some are even free and others are sold by companies specializing in this area. Some Visual Basic programmers may remember the early days when VBX controls were the staple of the controls world. All that’s changing rapidly with .NET.

Visual Studio Designer Layout Tools

Adding Controls Without Using the Designer

TextBox and Label Controls

The TextBox control is the most common control used for entering typed data into a form or dialog. It is familiar to everyone. The basic TextBox control is very simple to use. You just drag it from the toolbox to your form in designer view. You can use the resize handles on the edges to make it any size you want and drag it to position it on your form. The default TextBox is a single line. You can set the Multiline property to true if you want your control to accept newlines. The default is false. Where do the newlines go? That depends on whether the form has a default button set up with the AcceptButton property. If it doesn’t then the newline is just ignored. I will discuss this more thoroughly when we cover dialogs.

Exchanging information with a TextBox control is extremely easy. The Text property is used to both get and set the text in the control. Unfortunately the only data type we can use is a String object. If we want to have the user enter an integer, for example, we need to convert the character string to and integer. As you will see, this isn’t that difficult, but there are a few wrinkles we need to cover.

A Label control is in some ways similar to a TextBox except that it can only be used for output and not input. Label controls are most often used to place static text on the form or dialog. This is different than painting it with DrawString. The text is encapsulated in a control and not drawn on the client area directly.
Rather the control is drawn by .NET. What’s very neat about the Label control is that the text can be changed at run time. Just change the Text property. The Label control has no borders and the background is that of the form so it looks just like you painted the text with DrawString. You can change the background color with the BackColor property. The default is set from the background color of the control’s parent, which in this case is the form.

Figure 4-1 shows a simple form with a Label control, TextBox control, and a Button. The following program uses this form to demonstrate how you can convert a text string to an integer using the input from the TextBox textBox1:

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace Exception1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            int i;
            i = Convert.ToInt32(textBox1.Text);
        }
    }
}
```

![Form1](image)

**Figure 4-1**

The Convert class has many static methods that attempt to convert a character string to a target data type. The int data type is a 32-bit integer and we use the ToInt32 method for the conversion. The result is saved in the variable i. This all looks quite simple, doesn’t it? Unfortunately there is a pitfall. What if the user doesn’t enter an integer? Does Convert.ToInt32 merely ignore characters that
are not digits 0-9? Let’s find out. If we run the program without the debugger and enter "123abc456" in the TextBox control and click on the button we get the message dialog shown in Figure 4-2. What is an unhandled exception? Read on.

![Figure 4-2](image)

**Exceptions**

The above problem stems from the fact that `Convert.ToInt32` needs a way to communicate the fact that the character string we asked it to convert is not a valid integer. Many run-time environments use a special return value to indicate an error, for example, -1. But that doesn’t work here since -1 is a perfectly valid integer. Other techniques include using a Boolean parameter or return value to indicate that an error has occurred. A global variable is then set to reflect an error number. These are legacy techniques and still have merit, but more modern programming languages implement the concept of an exception. C++ includes this capability although most programmers don’t use it since the run-time does not report errors with exceptions.

Exceptions are very simple to use. The following is the structure of an exception handler:

```c++
try
{
    //code to be executed
}
catch (Exception e)
{
    //your exception handler.
}
finally
{
    //optional code to be executed whether or not
    //an exception is generated.
    //This block is optional.
}
```

The code to be executed subject to the exception handler is placed in the braces following the `catch` keyword. If an exception is generated then control is immediately transferred to the code inside the `catch` block. You should note that no code is executed after the code statement that caused the exception.
All exceptions in .NET are derived from the base class `Exception`. Each method in the .NET FCL that generates one or more exceptions is documented with the particular class of the exception. For example, the `Convert.ToInt32` method will generate a `FormatException` if you do not enter the characters for a valid integer. You can catch all exceptions by using the `Exception` class in the `catch` block. If you want to only catch a specific exception you can use the proper class name in the `catch` block. In fact you can have more than one catch block to differentiate between multiple exception types. That's a more advanced technique and I will defer discussion right now. If all you want to do is test the success of a method call you can use the `Exception` class here.

The finally block is used to make sure that a block of code is executed regardless of whether the exception is generated (thrown.) Usually this block is used for cleanup purposes such as closing an open file or resetting other values. The `finally` block is optional and you can just eliminate it if you want.

The following example demonstrates the use of an exception handler as well as changing the text in a `Label` control. If we have not entered an integer, or the value entered is not valid, this label is set to "unknown." Here is the code:

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace Exception2
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            int i=0;
            try
            {
                i = Convert.ToInt32(textBox1.Text);
                label3.Text = i.ToString();
            }
            catch (FormatException fe)
            {
                MessageBox.Show(fe.Message);
                label3.Text = "undefined";
            }
            finally
            {
                textBox1.Text = string.Empty;
            }
        }
    }
}
```
The exception handler is specific for the `FormatException`. To alert the user that he/she has entered an improperly formatted string I pop up a message box using the `MessageBox` class. This message box displays the text description for the exception. The `static` method `Show` has several overloads the simplest of which just takes a message string. I also set the label control to "unknown." To demonstrate the use of the `finally` block I am forcing the text box to be cleared. `String.Empty` is a static property of the `String` class that evaluates to a zero length string. Figure 4-3 shows what the form looks like with the integer 1234 entered and Figure 4-4 and Figure 4-5 show the result of entering an illegal string such as "1234x."

![Figure 4-3](image1)

**Figure 4-3**

![Figure 4-4](image2)

**Figure 4-4**
Are we home free? Not quite. A careful examination of the `ToInt32` method shows that another exception is thrown, an `OverflowException`. This exception is generated if an integer that is less than -2,147,483,648 or greater than 2,147,483,647. This is the range of a signed 32-bit integer. Unfortunately we are not catching this exception and we will get a message similar to Figure 4-2. There are two approaches to handling this. We could add another `catch` block for this exception and do something appropriately, or we could change the exception class in the `catch` block to the `Exception` class which would catch all exceptions. In this case the second approach is probably best since the user will know what the error was in the text passed to the `MessageBox` which is shown in Figure 4-6.

![Figure 4-5](image)

**Check Boxes**

To demonstrate how to use check boxes lets create a simple application to format a line of text using a combination of the bold, italic, and underline font styles. Create the form shown in Figure 4-7 by following these steps:

1. Place a `Label` control on the form with the text "Enter some text:".
2. Place a `TextBox` control to the right of the `Label` control. Change its name to "unformattedText" using the properties window.
3. Place three `CheckBox` controls on the form and change the control text to "Bold", "Italic", and "Underline" using the properties window.
4. Change the three `CheckBox` controls names to "boldCk", "italicCk", and "underlineCk" respectively again using the properties window.
5. Add two side by side Label controls with the text "Your formatted text:" and "formattedText".
6. Change the name of the lower right Label control to "formattedText".

At this point you can build your project and run the application. You can check and uncheck the check boxes and enter text in the text box control.

![Text Formatter](image)

**Figure 4-7**

All we need to do is format the text in accordance with the selections you make with the check boxes and display it in the lower right label control. The first thing that might come into your mind is to add a button and place the appropriate code in an event handler for its Click event. That would work just fine. Instead of doing it that way I can show you a nice feature of the TextBox control. We can add an event handler for the TextChanged event of the TextBox control. Every time any change is made to the text in the control this event is triggered. If we use this technique the formatted text can be made to display the instant something is typed and it continues to change for every character typed. Add this event handler using the properties window as shown in Figure 4-8.
Here is the complete program:

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

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

        private void unformattedText_TextChanged(object sender, EventArgs e)
        {
            FontStyle myStyle = new FontStyle();
            if (boldCk.Checked) myStyle |= FontStyle.Bold;
            if (italicCk.Checked) myStyle |= FontStyle.Italic;
            if (underlineCk.Checked) myStyle |= FontStyle.Underline;

            Font myFont = new Font("Arial", 12, myStyle);
            formattedText.Font = myFont;
        }
    }
}
```
I discussed the FontStyle enumeration in Chapter <ref>. All we need to do is to set the myStyle variable to the combination of the selections we have checked. Remember that we can combine these styles using a bitwise OR operator. Be sure to use the new operator to create the initial style or the program will not compile because the |= operator would be attempting to use an uninitialized variable. The Checked property of the CheckBox control is used to determine if a given check box is checked. All we do then is to set the style from the FontStyle enumeration. We create a new Font object and use it as the Font property of the unformattedText Label control. Copying the unformatted text from the text box to the label actually outputs the formatted text. Be sure to call Dispose on the font you created before returning from the event handler. Figure 4-9 shows a sample of our program. When you run the program you will see that the formatted text changes as you type.

Figure 4-9

There is one minor refinement that we can make without writing a single line of code. If you run the program as I have described you will notice that if you check or uncheck a check box that the new style is not reflected in the formatted text immediately. If you type some new characters or change the text box in any way then all of a sudden the formatted text changes. Wouldn't it be nice if we could change the formatted text the instant we changed the state of any of the three check boxes. I am sure it doesn't surprise you that the CheckBox control has a CheckedChanged event that is triggered when the state of the check box is changed. What might surprise you is that you don't need to create a new event handler for this event. All we need to do is use the drop down in the properties
window and select the event handler that we created for the **TextBox** control. Figure 4-10 shows what the properties window should look like when you do this. Repeat this for the remaining two check box controls and you are all set. Now the application will change the formatting of the text the instant you click on any one of the three check boxes.

![Properties Window](image:properties.png)

**Figure 4-10**

**Radio Buttons**

Radio buttons are very similar to check boxes. In addition to their appearance the big difference is that only one of a group of radio buttons can be checked at a time. If we add radio buttons to a form, all the radio buttons become part of the same group. If you wish to have more than one group of radio buttons you will need to place them in a group box or a panel. I will discuss panels and group boxes a little later in this chapter.

A simple example demonstrates the basic functionality of radio buttons. We will create a simple application with three radio buttons that result in changing the background color of the form. Start by dragging three buttons from the toolbox and aligning them appropriately. Use the properties window to change the **Text** properties of the buttons to "Red", "White", and "Blue". Also change the names to "redButton", "whiteButton", and "blueButton". Your designer view should look like Figure 4-11.
Now add an event handler for the `CheckChanged` event for the Red button. Set the `CheckChanged` property of the remaining two buttons to be handled by the event handler for the Red button. Here is the code you need to enter to implement the example:

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace Radio1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
            this.whiteButton.Checked = true;
        }

        protected override void OnPaint(PaintEventArgs e)
        {
            if (redButton.Checked) this.BackColor = Color.Red;
            if (whiteButton.Checked) this.BackColor = Color.White;
            if (blueButton.Checked) this.BackColor = Color.Blue;
        }

        private void redButton_CheckedChanged(object sender, EventArgs e)
        {
            Invalidate();
        }
    }
}
```
The implementation is straightforward. The constructor is used to make sure that the desired button is initially checked. If you don't do this then the first button in the tab order is checked. I will discuss tab order in the chapter on dialogs. The `OnPaint` method merely checks which one of the three buttons is checked and sets the background accordingly. I ensure that the form will be repainted by calling the `Invalidate` method in the event that is invoked by all three buttons. A slight performance improvement might be achieved by the use of `else` clauses instead of always checking the three buttons. As soon as one button is found to be checked the `OnPaint` method would then return. Figures Figure 4-12 and Figure 4-13 show the output when White and Red are checked respectively.

![Figure 4-12](image1)

![Figure 4-13](image2)
**List Boxes**

The `ListBox` control is a very common control used to let the user pick from a list of items. In many respects a list box is similar to a set of check boxes or radio buttons. Single or multiple items can be selected from a list box depending on how it is configured. The contents of a list box can be static and specified using the designer, or the contents can be changed programmatically. In fact it is very easy to bind the contents of the list box to an object or even a table in a database.

Let's start by creating a `ListBox` control with the days of the week as its contents. Drag the control from the toolbox to your form in designer view. You can worry about sizing it after we add the contents. Make sure it is selected and go to the properties window. Find the `Items` property. Figure 4-14 shows the properties window. The `Items` property is a reference to a `ListBox.ObjectCollection`. This collection contains the items to be displayed in the list box. If you click on the ellipses icon you open a special collections editor that allows you to set up a collection of text strings to be displayed in the list box. Figure 4-15 shows the editor with the days of the week entered. You can always come back to this editor in the same way to update your selections.

![Properties window](image)

**Figure 4-14**
You can now resize the control to fit all the selections or only some of them. If you make the control too small you will get a vertical scroll bar so the user can look at all the contents. If you want to shrink the list box horizontally you will need to set the HorizontalScrollBar property to true if you want a horizontal scroll bar as well.

In order to demonstrate the use of our list box let's add a button and a text box control. The objective will be to display the selected item in the list box in the text box. There are numerous ways to determine what item has been selected. The simplest is to use the SelectedItem property of the ListBox class. This property is set to the object that corresponds to the selection. We can use its ToString method to convert it to text. In fact, the text used in the list box is obtained by using the ToString method on each of the objects in the Items collection. In fact, any objects can be added to the Items collection as long as there is a meaningful string representation returned by its ToString method.

There is one minor issue to consider. What if no selection has been made? In this case the SelectedItem property evaluates to null. If you try to call ToString you will cause an exception. You can certainly use a try/catch construct to handle this. I don' advocate that technique. Two other solutions are much better. We can make sure there is always a selected item by setting the SelectedItem property to a valid object. That's possible, but we can do it an easier way. We can set the SelectedIndex property to the zero offset index of the desired item. This in turn sets the SelectedItem property. The value of the SelectedItem property is -1 if no item is selected. The second solution, if we don’ want to have a default selection, is to test the SelectedIndex property for -1. Here is our example whose form is shown in Figure 4-16:

```csharp
ListBox1
```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace ListBox1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
            // uncomment the next line for a default selection
            // listBox1.SelectedIndex = 0;
        }

        private void button1_Click(object sender, EventArgs e)
        {
            if (listBox1.SelectedIndex != -1)
                outBox.Text = listBox1.SelectedItem.ToString();
        }
    }
}

Figure 4-16

The ListBox.ObjectCollection class has an Add method that can be called to add items to the collection. The following program works exactly as the previous one:

ListBox2
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Windows.Forms;

_SampleCode_
using System.Text;
using System.Windows.Forms;

namespace ListBox2
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
            listBox1.Items.Add("Sunday");
            listBox1.Items.Add("Monday");
            listBox1.Items.Add("Tuesday");
            listBox1.Items.Add("Wednesday");
            listBox1.Items.Add("Thursday");
            listBox1.Items.Add("Friday");
            listBox1.Items.Add("Saturday");
            listBox1.SelectedIndex = 0;
        }

        private void button1_Click(object sender, EventArgs e)
        {
            if (listBox1.SelectedIndex != -1)
            {
                outBox.Text = listBox1.SelectedItem.ToString();
            }
        }
    }
}

Binding Controls to Objects

Many controls can get their data from objects that support binding. For a control to support binding it must implement the IList interface. I will cover interfaces in Chapter <ref>. For now all you really need to know is that the IList interface specifies a set of properties and methods that must be implemented if the object supports the interface. The IList interface properties and methods are used by controls to obtain the objects to be added to the Items collection. The abstract class Array, from which all C# arrays are derived from, supports the IList interface. That means that we can bind to an array of any object. Here is some code that binds a list box to an array of integers:

ListBoxBind
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
namespace ListBoxBind
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
            listBox1.DataSource = iarray;
        }
        private int[] iarray = {1,2,3,4,5,6,7,8,9,10};
    }
}

Figure 4-17 shows the output for this program. You can set the \textit{DataSource} property using the properties window rather than placing the code in the constructor. What is really the advantage of the \textit{DataSource} property is that it can bind to objects such as \textit{DataSets} that are associated with tables in databases. I will discuss this in Chapter <ref>.

![List Box Binding Example](image)

Figure 4-17

**Panels**

The \textit{Panel} control is very similar to a form. The default panel is borderless and has no caption (title) bar. Panels are normally used to group and contain other controls. You can also paint directly to the panel by using the \textit{Paint} event. You start by dragging the \textit{Panel} control from the toolbox and sizing it to your needs. You can then drag other controls into the panel.

The \textit{Panel} control supports the \textit{AutoScroll} property and if you set it to \textit{true} you will get scroll bars if the size of the panel is too small to display all the controls it contains. Two other useful properties are quite powerful. The \textit{Enable} property if set to \textit{false} will disable all the control inside the panel. They are still displayed, but grayed out and are non-functional. This comes in handy when you want to disable a group of controls depending on the program's current state. The \textit{Visible} property can be set to \textit{false} to completely hide the panel and all its
controls. The Visible property is found on all controls, but using a panel allows you to hide a whole group of controls in one operation.

To demonstrate create a form that looks like Figure 4-18 using the designer and the toolbox. Set the Text properties of the controls as shown. Rename the three buttons to "hide", "disable", and "restore" using the properties window. Double click each to add event handlers.

Enter the following code for each event handler:

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace Panel1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void panel1_Paint(object sender, PaintEventArgs e)
        {
            Graphics g = e.Graphics;
        }
    }
}
```
g.DrawString("hello", Font, Brushes.Black, 10, 10);

private void hide_Click(object sender, EventArgs e)
{
    panel1.Visible = false;
}

private void disable_Click(object sender, EventArgs e)
{
    panel1.Enabled = false;
}

private void restore_Click(object sender, EventArgs e)
{
    panel1.Visible = true;
    panel1.Enabled = true;
}

Figure 4-19 shows the form with the panel enabled and visible. Figure 4-20 shows the panel hidden and Figure 4-21 show the panel disabled. When you run the program try to enter something in the text box while the panel is grayed out. You won't even be able to give the focus to the TextBox control no less enter anything.
Scrolling With a Panel

A panel provides a very easy way to scroll without the need to adjust to position of everything you paint to accommodate the current scroll position. All you do is to place a panel in your form and set its size to the size needed to
accommodate all of your output. You need to make sure the AutoScroll property of your form is set to true. This scrolls the panel within the form and NOT the panel itself. That's different than setting the AutoScroll property of the panel. Remember, that won't automatically scroll anything drawn with the Graphics object. Start by placing a panel in the form with the appropriate position for the upper left hand corner. If you know the size in advance you can then set the size in the properties window. The example to follow uses a fixed size for the width of the panel but adjusts the height programmatically. Figure 4-22 demonstrates this step.

![Scrolling a Panel](image)

Figure 4-22

The following code outputs the first 100 integers using a 16 point Arial font. I added a Paint event handler for the form in the usual way. Note, you can't override the OnPaint method of the Panel class since we have no class derived from Panel in which to do this. I have also set the AutoScroll property of the form. To determine the vertical scaling for each call to DrawString I am using the Height property of the Font class. Figures Figure 4-23 and Figure 4-24 show the ability to scroll over the entire range of the output. You should note that when we scroll down that the text flows up to the top edge of the form's client area. That's because we are scrolling the panel within the form and not the content of the panel.

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

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

    private void panel1_Paint(object sender, PaintEventArgs e)
    {
        const int nLines = 100;
        Font font = new Font("Arial", 16);
        int cy = font.Height;
        panel1.Height = nLines * cy;
        Graphics g = e.Graphics;
        for (int i = 0; i < nLines; ++i)
            g.DrawString(i.ToString(), font, Brushes.Black, 0, i * cy);
        font.Dispose();
    }
}
```
Docking a Panel

The Panel control inherits the Dock property from the Control class. Theoretically any control derived from Control can be docked. However, docking controls such as buttons or check boxes would rarely make sense. Docking causes a control to be aligned to one or more edges of the parent control and automatically sized to fill that edge. Table 1 shows the possibly values that can be applied to the Dock property. Figure 4-25 shows a panel docked to the left edge of the form. I changed the background color of the panel so you could see the area the panel covers. If you resize the form, the panel will always fill the form vertically and its width will remain constant. This works the same for the other three edges.

<table>
<thead>
<tr>
<th>Member Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>The control is not docked.</td>
</tr>
<tr>
<td>Top</td>
<td>The control's top edge is docked to the top of its containing control.</td>
</tr>
<tr>
<td>Left</td>
<td>The control's left edge is docked to the left edge of its containing control.</td>
</tr>
<tr>
<td>Bottom</td>
<td>The control's bottom edge is docked to the bottom of its containing control.</td>
</tr>
<tr>
<td>Right</td>
<td>The control's right edge is docked to the right edge of its containing control.</td>
</tr>
<tr>
<td>Fill</td>
<td>All the control's edges are docked to the all edges of its containing control and sized appropriately.</td>
</tr>
</tbody>
</table>
Figure 4-25

The Fill DockStyle is a very interesting option. It causes the control to completely fill its parent control (container). As we resize the form, for example, the panel would resize to always fill the client area. It might not seem that this is a useful option since the panel would essentially act like the client area of the form itself. However, there are cases where the Fill docking style can be very useful. If we have more than one panel in the form we can dock one panel to the bottom and another panel to fill. The panel that is set to fill will fill the remaining area of the form and not overlap the panel docked to the bottom. In other words, the filled panel docks to the top edge of the panel docked to the bottom of the form. You can use the AutoScroll property these panels as well.

Tab Index and Tab Stop

<TBA>