Basic Controls

Label Controls

- Rather than paint text onto your form you can use the *Label* control.
- Set the *Text* property to the text you want displayed.
- It will be formatted to fit the rectangle you define. Need to set AutoSize to false to use rectangle.
- Similar to using *DrawString* with a rectangle.
- A label's text can be changed at runtime so they make nice placeholders for dynamically generated text.

TextBox Control

- The *TextBox* is the familiar control you use to enter both text and numeric data.
- No automatic conversion is performed for numeric data.
- Use the *Convert* class to convert from one type to another.
- Careful – bad data will generate an *exception*.

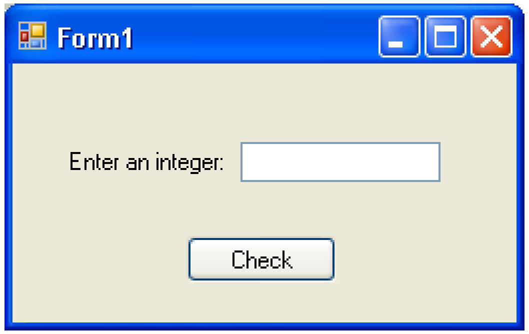
Example Without Exception Handler

```
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
    {
```

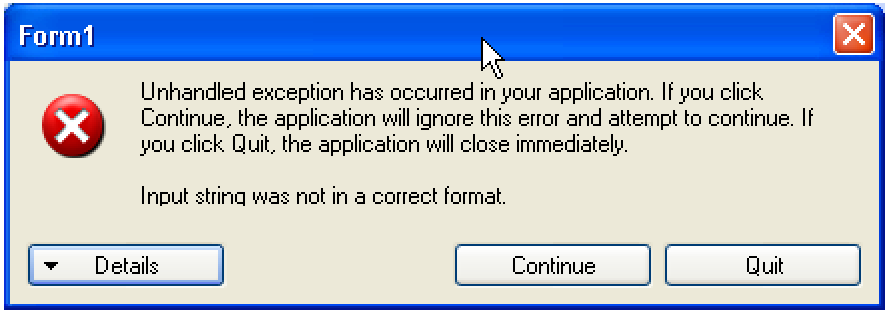
Example Without Exception Handler – contd. (Exception1)

```
        public Form1()
        {
            InitializeComponent();
        }
        private void button1_Click(object sender,
EventArgs e)
        {
            int i;
            i = Convert.ToInt32(textBox1.Text);
        }
    }
}
```

The Form



Bad Data Exception



Unhandled exception has occurred in your application. If you click Continue, the application will ignore this error and attempt to continue. If you click Quit, the application will close immediately.

Input string was not in a correct format.

Exceptions Syntax

```
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.
}
```

Revised Example (Exception2)

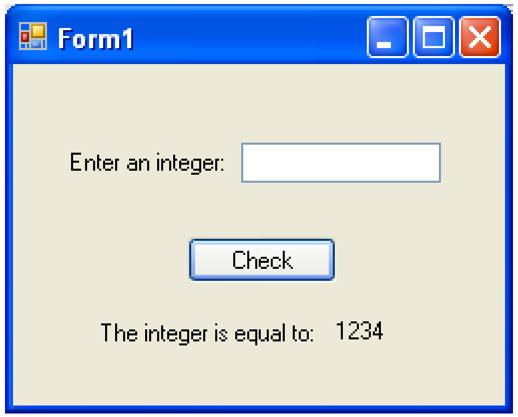
```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
namespace Exception2
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        private void button1_Click(object sender, EventArgs e)
        {
```

Revised Example – contd.

```
                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;
                }
            }
        }
}
```

Pops up a standard message box.

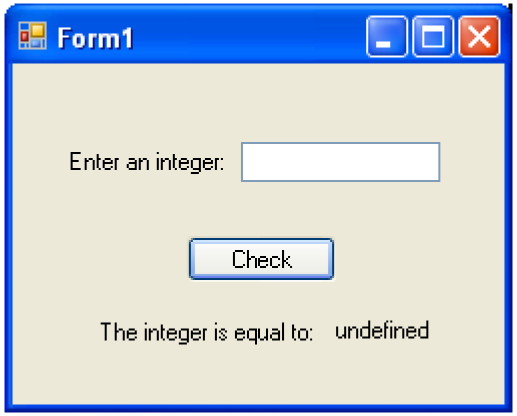
With Correct Input



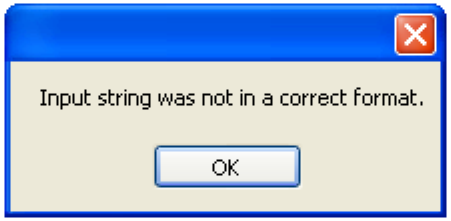
Enter an integer:

The integer is equal to: 1234

Catching the Exception



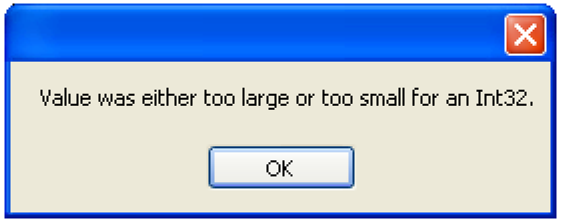
The integer is equal to: undefined



Input string was not in a correct format.

Overflow Exception

- This exception is generated if an integer that is less than -2,147,483,648 or greater than 2,147,483,647.
- You can have multiple exception blocks to distinguish between different exceptions for the same method call.

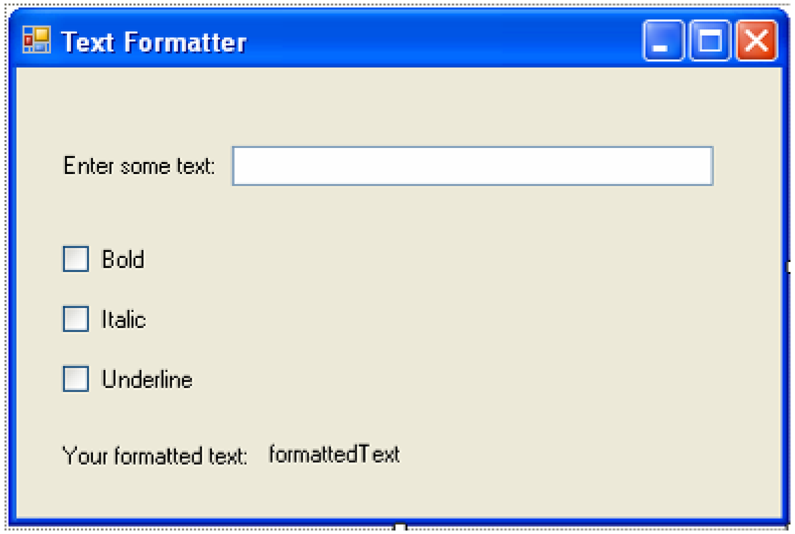

Value was either too large or too small for an Int32.

Possible message box for this error.

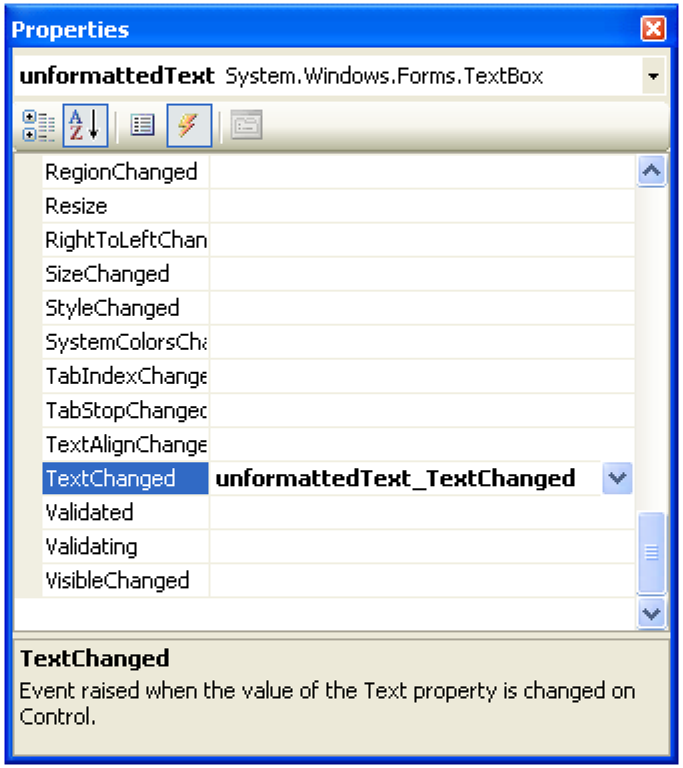
Check Boxes - Example

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".

The Form


Text Formatter
Enter some text:
Bold
Italic
Underline
Your formatted text:
formattedText

TextChanged Event Handler


Properties

unformattedText System.Windows.Forms.TextBox

RegionChanged	
Resize	
RightToLeftChan	
SizeChanged	
StyleChanged	
SystemColorsCha	
TabIndexChange	
TabStopChangec	
TextAlignChange	
TextChanged	unformattedText_TextChanged
Validated	
Validating	
VisibleChanged	

TextChanged
Event raised when the value of the Text property is changed on Control.

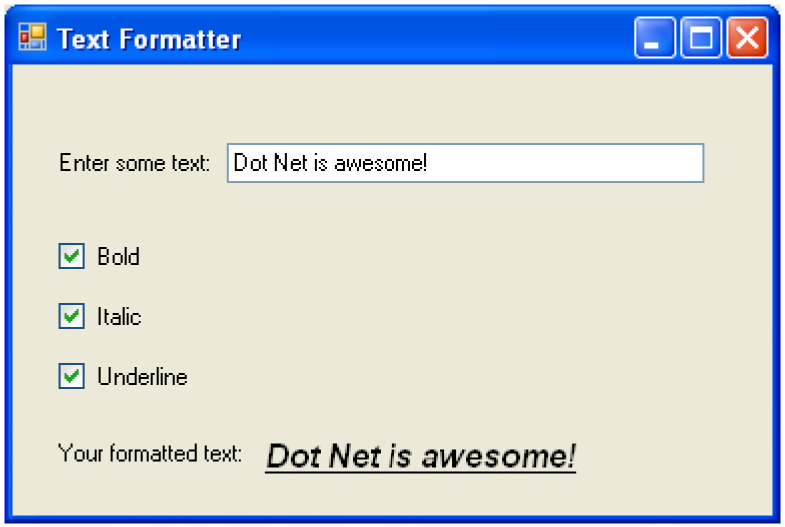
The Code (Checkboxes)

```
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();
```

The Code – contd.

```
            }
            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;
                  formattedText.Text = unformattedText.Text;
                  myFont.Dispose();
          }
     }
}
```

The Result



Text Formatter

Enter some text: Dot Net is awesome!

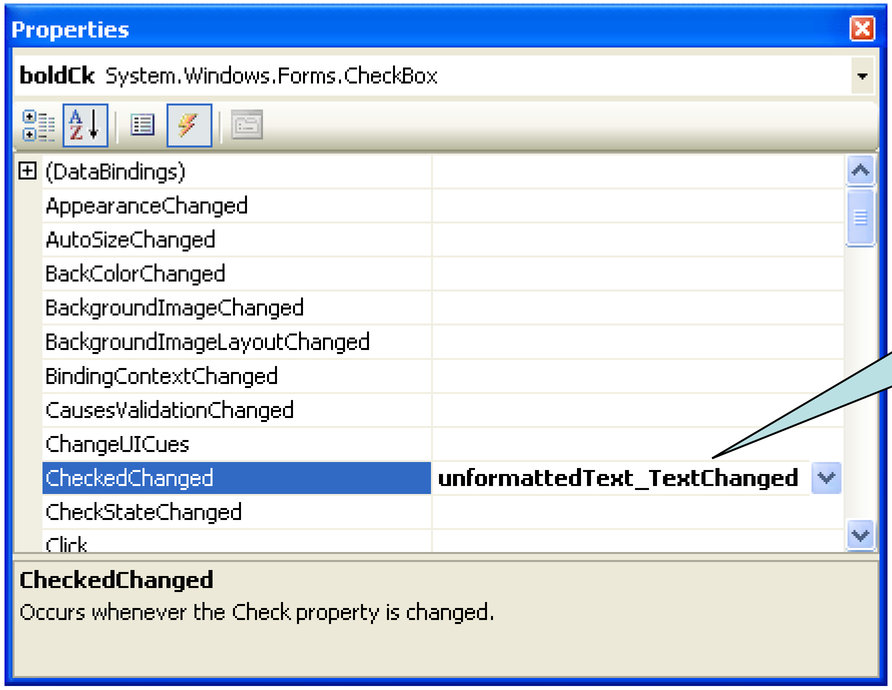
[✓] Bold

[✓] Italic

[✓] Underline

Your formatted text: ***Dot Net is awesome!***

CheckedChanged Property



Properties

boldCk System.Windows.Forms.CheckBox

(DataBindings)	
AppearanceChanged	
AutoSizeChanged	
BackColorChanged	
BackgroundImageChanged	
BackgroundImageLayoutChanged	
BindingContextChanged	
CausesValidationChanged	
ChangeUICues	
CheckedChanged	unformattedText_TextChanged
CheckStateChanged	
Click	

CheckedChanged
Occurs whenever the Check property is changed.

Set to same handler as the text box.

Now the text changes when we check or uncheck a style.

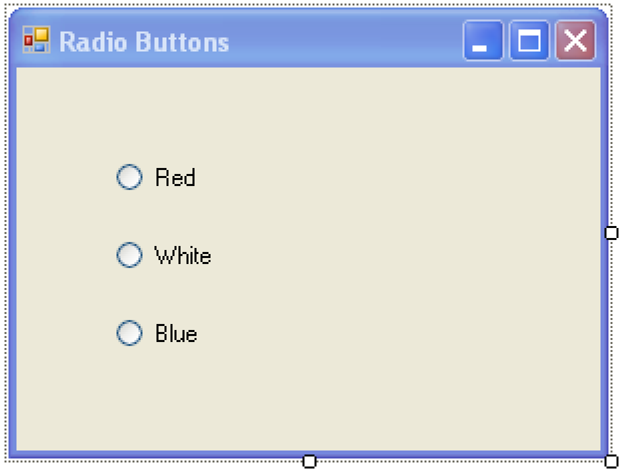
Radio Buttons

- 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".
- Change the names to "redButton", "whiteButton", and "blueButton".
- 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.

Radio Button Notes

- 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 discus panels and group boxes a little later in this chapter.

The Form



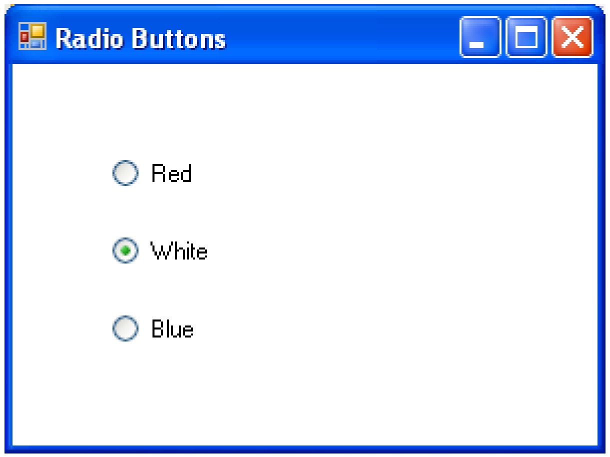
Radio Button Example

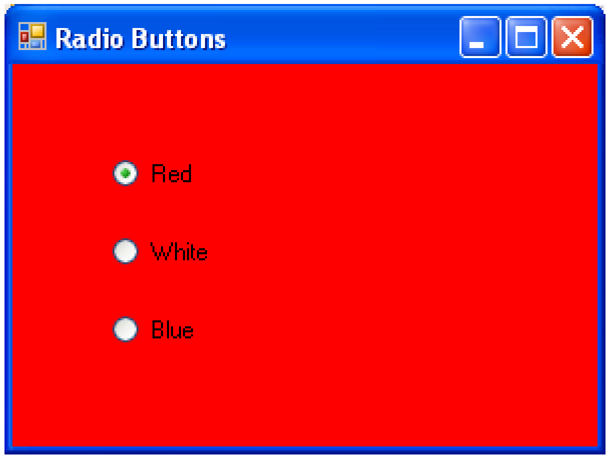
```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
namespace Radio1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
```

Radio Button Example – contd.

```
            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 Output

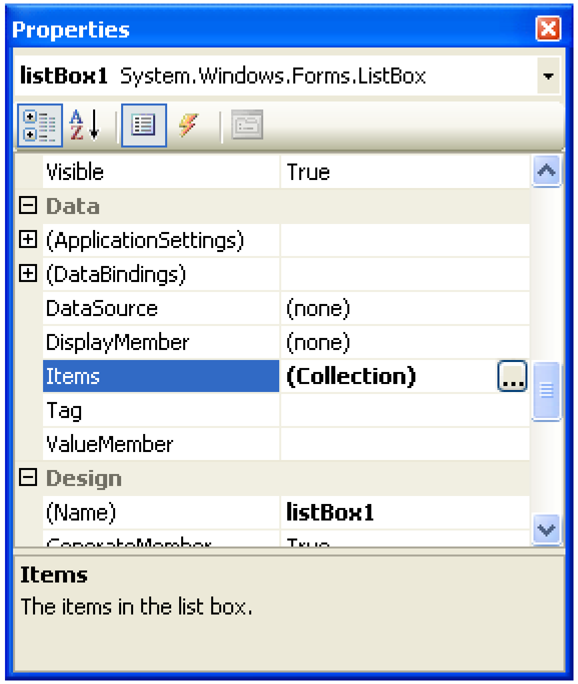




List Boxes

- The list box control allows you to provide a list of selections for the user.
- An option allows multiple selections, but the default is a single selection.
- You can populate the list box with static elements or dynamically at execution time.
- It is also very easy to bind this control to a database. (Covered later in the book.)

Adding the List Items



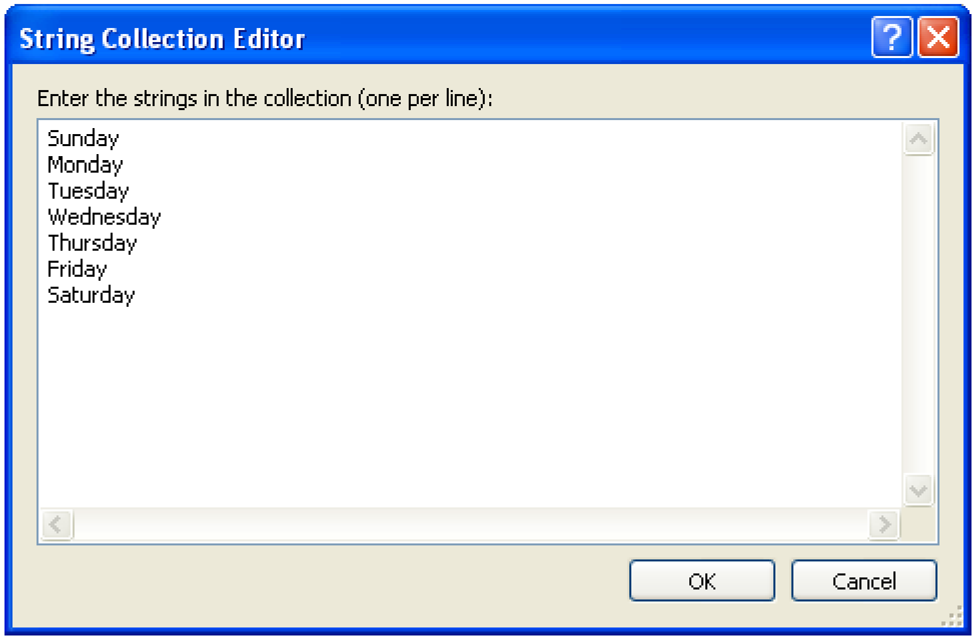
Properties

listBox1 System.Windows.Forms.ListBox

Visible	True
⊟ Data	
⊞ (ApplicationSettings)	
⊞ (DataBindings)	
DataSource	(none)
DisplayMember	(none)
Items	**(Collection)**
Tag	
ValueMember	
⊟ Design	
(Name)	**listBox1**

Items
The items in the list box.

Adding the List Items – contd.



Enter the strings in the collection (one per line):

```
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
```

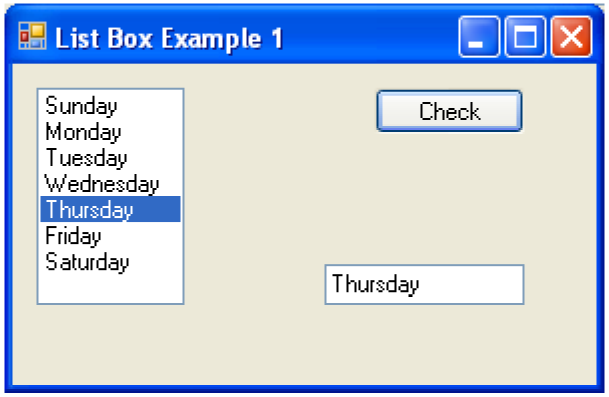
Simple Example

```
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
    {
```

Simple Example – contd.

```
        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();
        }
    }
}
```

Result


List Box Example 1
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Check
Thursday

Populating the Control at Execution Time

```
ListBox2
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
m.Drawing;
using System.Text;
using System.Windows.Forms;
namespace ListBox2
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
```

Populating the Control at Execution Time – contd.

```
            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

- Any object that supports the *IList* interface can be used as a binding source for a list box control.
- All C# arrays implement *IList* since they are objects of the *Array* class.
- Obviously the data type must be a *string* or a data type that we can convert.
- For example, integers can be converted using the *ToString* method.
- Note – the list box control is clever enough to automatically call *ToString. .NET is neat!*

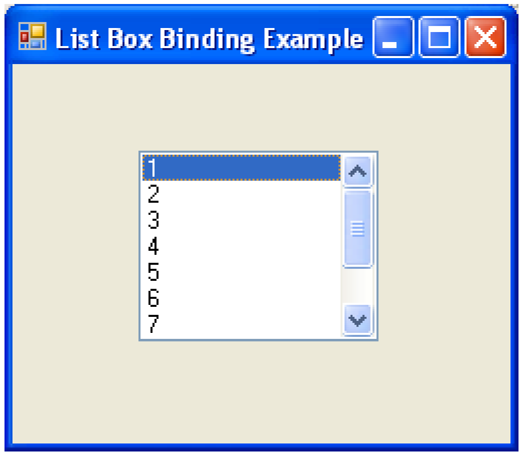
Binding to an Integer Array

```
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
{
```

Binding to an Integer Array – contd.

```
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
            listBox1.DataSource = iarray;
        }
        private int[] iarray = {1,2,3,4,5,6,7,8,9,10};
    }
}
```

Output

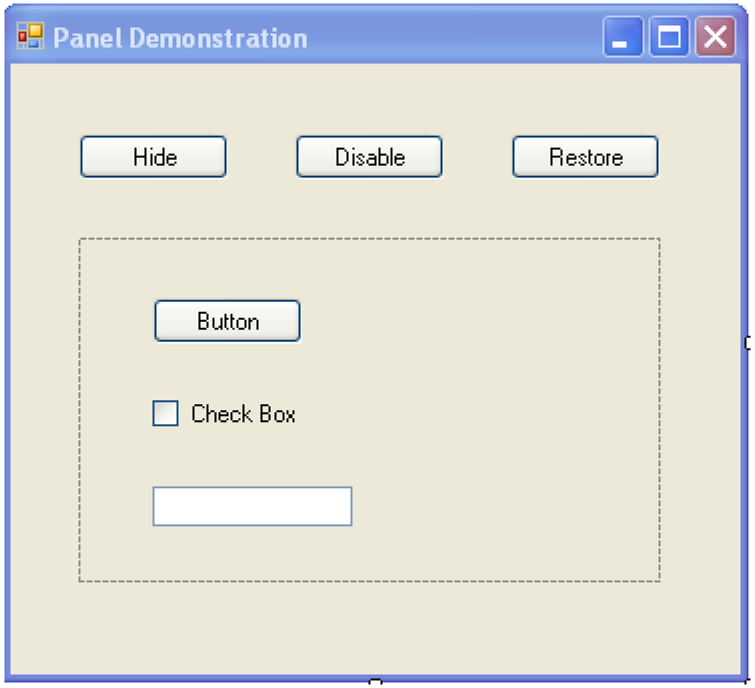


```
1
2
3
4
5
6
7
```

Panels

- Panels are very similar to forms.
- They are primarily used to contain other controls.
- They support the *AutoScroll* property.
- You can paint to a panel using its paint event.
- The *Visible* property can be used to hide a panel and its contents.
- The *Enable* property can be set to *False* to disable all controls inside the panel.

Example


Panel Demonstration
Hide
Disable
Restore
Button
Check Box

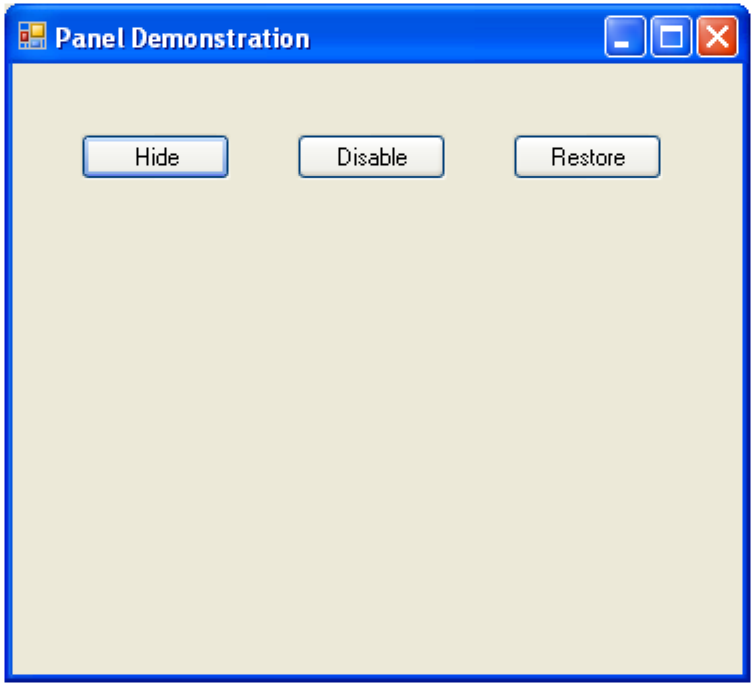
The Event Handlers

```
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)
        {
```

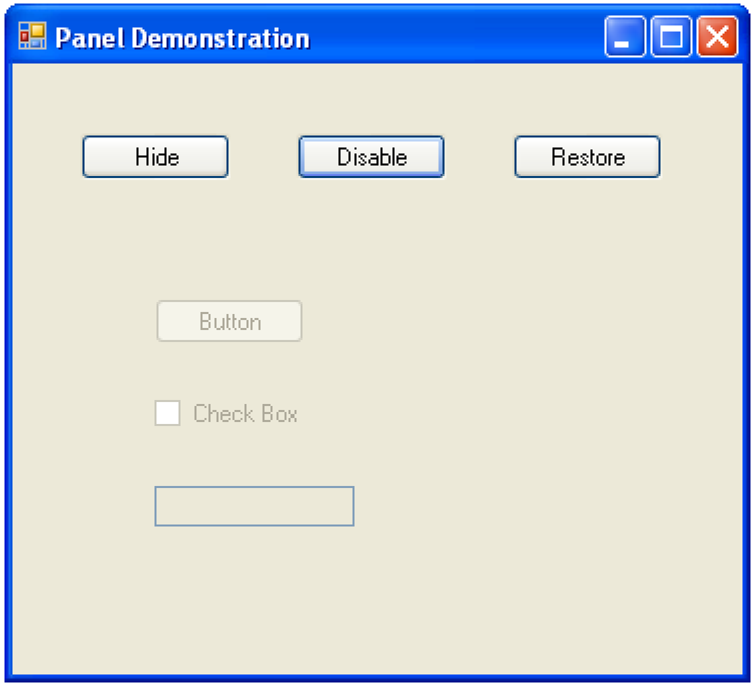
The Event Handlers – contd.

```
            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;
        }
    }
}
```

Hidden and Disabled Panel



Hidden



Disabled

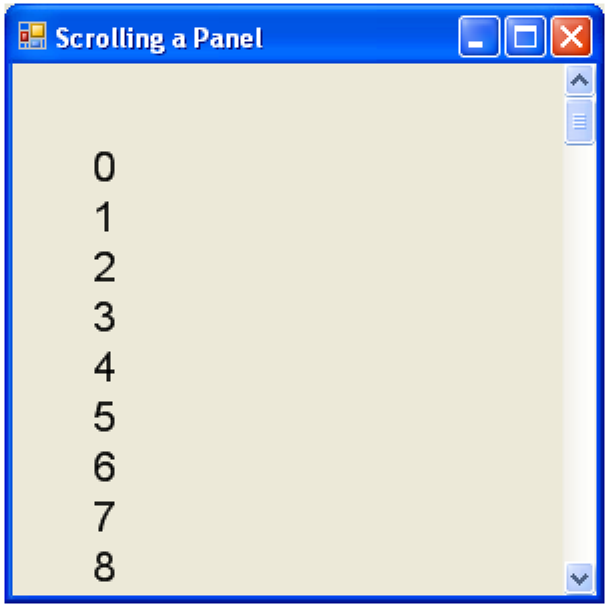
Scrolling a Panel and Using the Paint Event

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
namespace PanelScroll
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
```

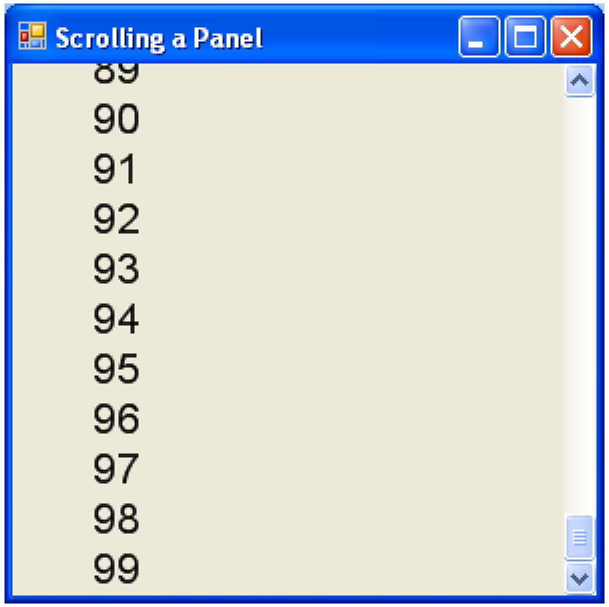
Scrolling a Panel and Using the Paint Event – contd.

```
        }
        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();
        }
    }
}
```

Output



```
0
1
2
3
4
5
6
7
8
```



```
89
90
91
92
93
94
95
96
97
98
99
```

Notes for Previous Example

- Don’t forget to set the *AutoScroll* property.
- You must use a paint event handler since you can’t override the *OnPaint* method since we have no derived class.
- Note that you don’t need to worry about the scroll position in your calls to the *Graphics* methods. The *Panel* control does all the work for you. Neat!

Docking a Panel

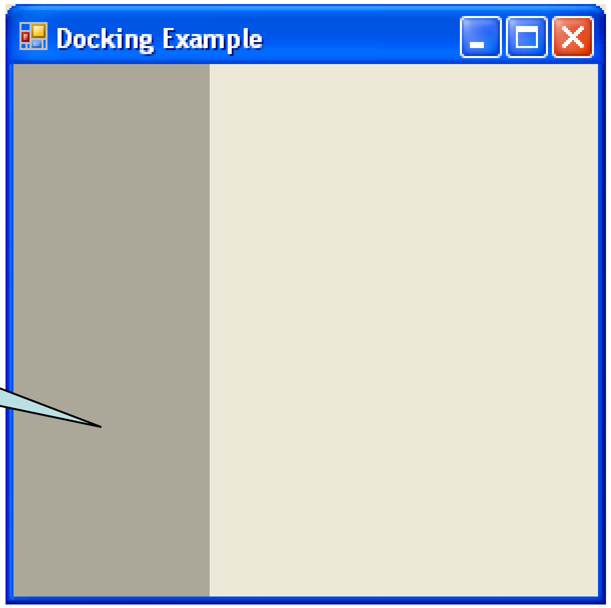
- The *Panel* control inherits the D*ock* property from the *Control* class.
- Use this property to lay out your forms.
- We will use this property later when I discuss *ToolStrips* etc.
- The following table shows the possible docking styles.

DockStyle Enumeration

Member Name	*Description*
None	The control is not docked.
Top	The control's top edge is docked to the top of its containing control.
Left	The control's left edge is docked to the left edge of its containing control.
Bottom	The control's bottom edge is docked to the bottom of its containing control.
Right	The control's right edge is docked to the right edge of its containing control.
Fill	All the control's edges are docked to the all edges of its containing control and sized appropriately.

Panel Docked to Left Side

The background color of the panel has been changed to show its position more clearly.


Docking Example