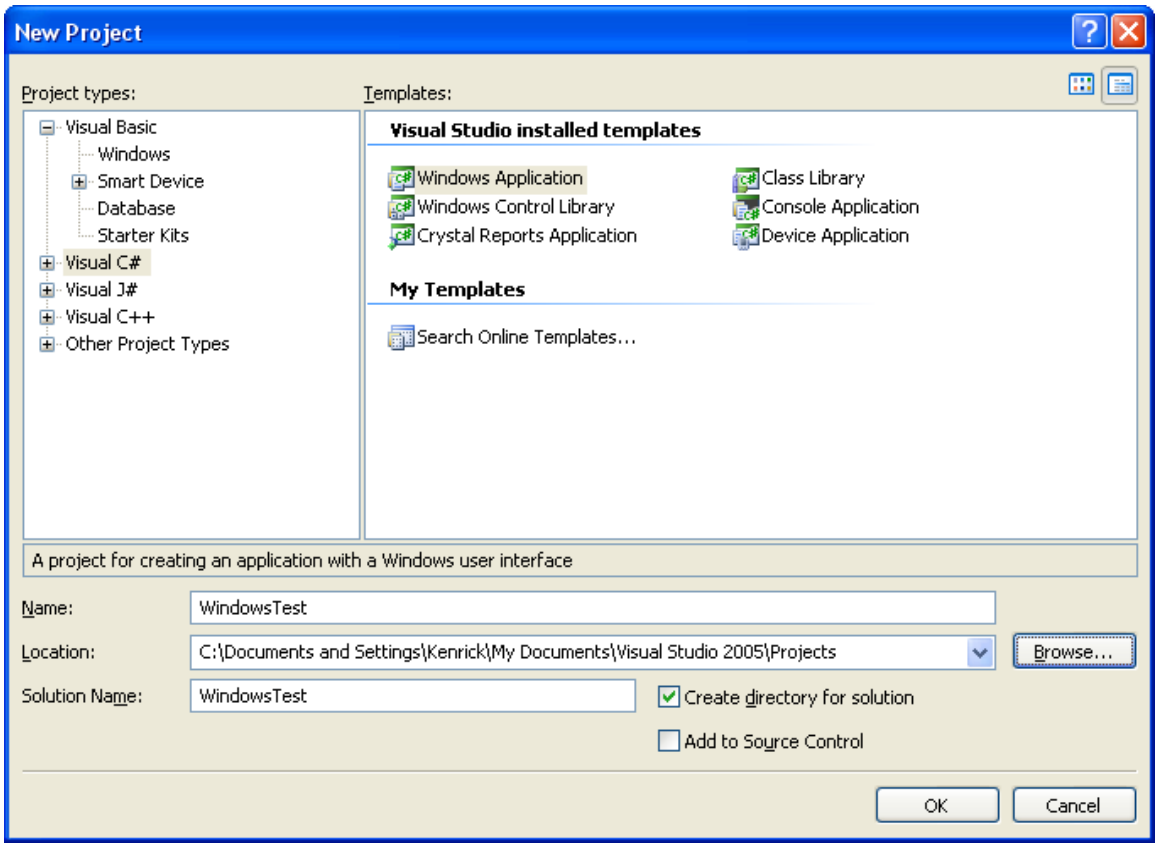
Creating Windows Forms Applications with Visual Studio .NET and C#
CS A331

Visual Studio .NET on a Windows platform gives you a multitude of classes to easily create typical Windows GUI applications. If you elect to use these features, currently a C# application will only run on a Windows machine. There is similar functionality for C# GUI applications under Mono. In this lecture we will present just the basics to create some simple Windows apps.

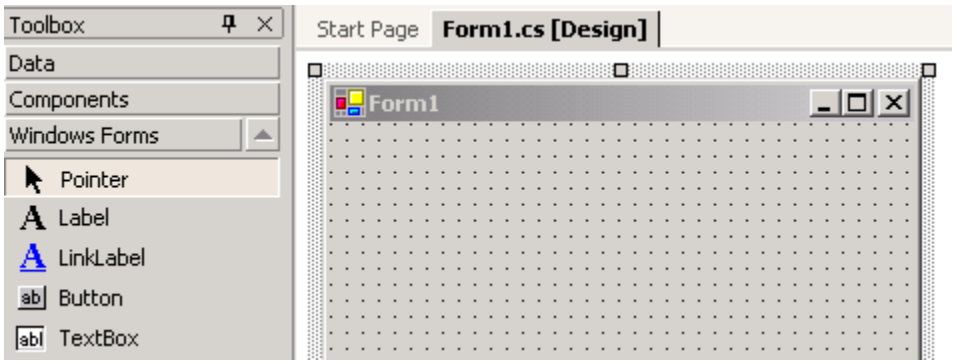
Although it is possible to create windows applications purely in a textual environment, we will instead make use of the Visual Studio .NET interface which lets us graphically drag and drop to determine the layout of our program. The .NET environment will then generate the necessary code for us to perform the layout.

The following screenshots were created in a previous version of VS .NET. If you have a different version, some windows and/or content may have changed but the basic process is the same. To create a Windows Forms Application, start Visual Studio .NET and create a new Visual C# Project. Make sure you select a Windows Application as the template.

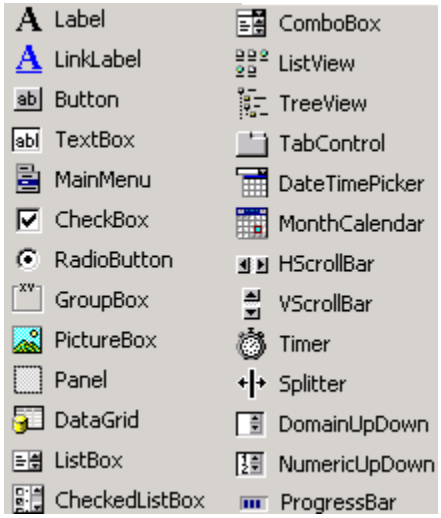


You will be presented with a blank **form**. This document assumes that you have the

Toolbox pane visible on the left of the screen and it is set to Windows Forms. This pane contains common **controls** which make up the elements on the form. The layout is shown below:



A control is a component with a visual representation. The `Control` class in the System.Windows.Forms namespace is the base class for the graphical objects on the screen. All controls are derived from the base control. Examples of controls include:

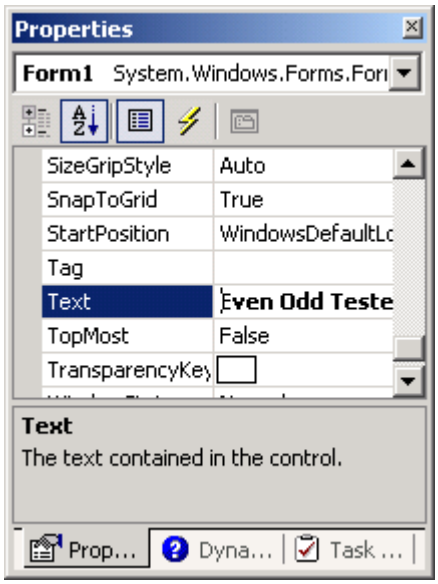


A `Form` is a type of `ContainerControl`. It can contain other controls and we can add controls to it.

In this document we will only discuss a few controls: Buttons, Labels, TextBox, ProgressBar, and PictureBox. A Button is just like it sounds, a button we can press and when it is pressed some code is invoked. A Label is a way to output some text on the form. A TextBox can be used to output data and provide a way for the user to enter text to the program (although a label is better for data that is purely output only). A ProgressBar displays a graphical bar of progress for some slow event. Finally, a PictureBox is used to load and display an image. Please refer to the .NET documentation for details about the other controls.

Let's make a simple Windows application that will allow the user to enter a number into a TextBox and press a button. Upon pressing the button we will output if the number is even or odd into a label.

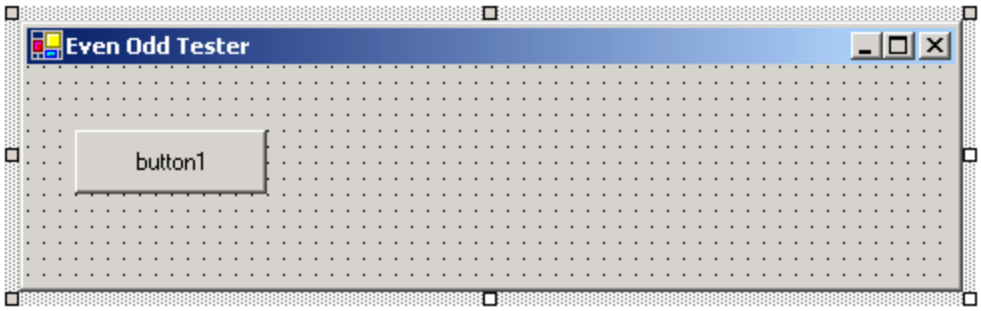
First, select the form by clicking on it. In the Properties Window, you can click and modify various properties of the form. By default, it is called `Form1`. Scroll through the properties; there are many available to change, such as the size, icon, width, locked, etc. One we can change is the title bar shown with the form. Here we change the title bar of the form to "Even Odd Tester" by changing the Text property.



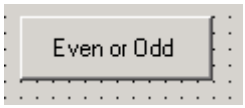
This changes the title of the form itself:

At this point you may wish to grab the "size" handles around the form, and resize it to the dimensions that you like.

To add controls to the form, select the control you want from the Toolbox on the left. Then click and drag in the form window in the location you want the control to go. If you select the Pointer tool, you can select existing items on the form and move them elsewhere. You can also click and resize the control on the form. Try dragging a button onto the form:



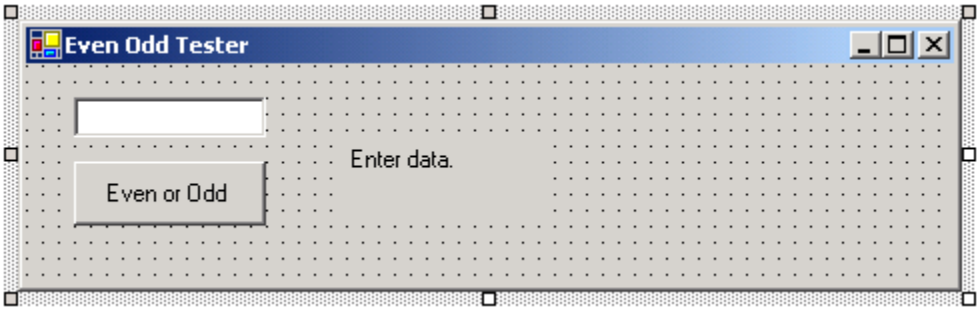
Click the button on the form and change its Name property to “buttonEvenOdd”. This is setting the name of the variable used to reference the button object from its default of “Button1”. Also change the button’s text property to “Even or Odd”. You should see the change to the text field take place on the form:



In the same fashion as the button, add a label to the form and name it “labelResult” and set its Text field to “Enter data”.

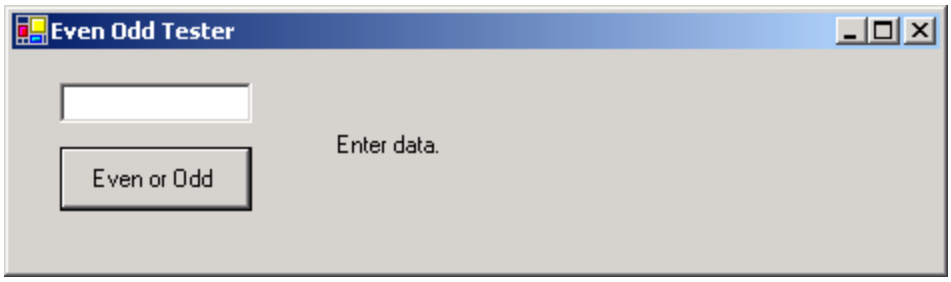
Also add a TextBox to the form, name it “textBoxData” and delete the contents of the Text field. As you add the controls to the form, you will see vertical lines that help you align controls to each other.

The result should look something like this:



At this point you should be able to execute the program you have created so far by clicking on the run button, shown as an arrow:

Your app should execute and look like the following:



You can enter data in the textbox and press the button. However, nothing will happen because no code is associated with these objects. We will add code to them shortly. A Windows application is based upon **event-based processing**. The application is waiting for some event to happen and reacts to those events rather than firing off sequential statements and exiting when they are done. At this point it may be useful to see the actual code that has been generated so far. By dragging our controls on the Form, Visual Studio has generated the code that would correspond to what is visually shown on the screen. You can look at the code at any time by right-clicking on the "Form1.cs" name and select "View Code" to see the C# code:

Form1.cs | Form1.cs [Design]

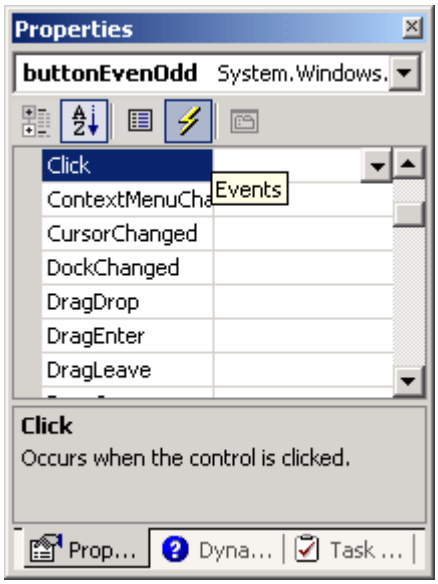
WindowsTest.Form1

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

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

You can see that Visual Studio has generated some code for you. There is more, but it's not generally made visible to the programmer.

To go back to the graphical form designer, click on the "Form1.cs [Design]" tab .

To add code behind an event, click the control you are interested in and then click on the events button (indicated by a lightning bolt) of the properties window. In this case, we would like to add an event to the button, so select the button on the form and then events:



The window shows all of the events supported by Windows. For example, the Click event is invoked when the button is clicked. The DragDrop event is invoked if an item is dragged and dropped on the button. You can scroll through the list to see the myriad of events.

Select the Click box and enter "buttonClicked". This is defining a method named "buttonClicked" that will be invoked when the button is clicked. Immediately Visual Studio will bring you to an empty code skeleton that it generates for the method:

```
private void btnEvenOdd_Click(object sender, EventArgs e)
{

}
```

The sender parameter is the object that invoked the event. In most cases, it is the actual object that was selected by the user, in this case, the button object that was clicked. The System.EventArgs parameter passes object-specific parameters to the event being handled. For example, in a drag-drop event this might indicate the object being dropped. We'll see an example of using the event args later. The buttonClicked() method is actually used as a Delegate for the System's Event Handler, but these details are somewhat hidden by the Visual Studio IDE.

Another, faster way to define an event is to simply double-click on the control in the design form. This will add code for a default event. For a button, this is click. The default event will vary for other controls.

For example, double-click on an empty part of the form itself:

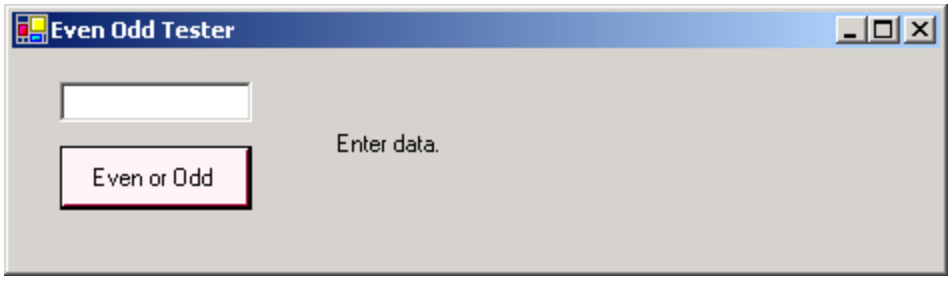
This adds the default event of Form Load:

```
private void Form1_Load(object sender, EventArgs e)
{

}
```

Any code we enter in here is executed when the form is loaded for the first time. In our case, this corresponds to when the application is initially started. Try adding the following code:

```
private void Form1_Load(object sender, EventArgs e)
{
  this.btnEvenOdd.BackColor = Color.LavenderBlush;
  MessageBox.Show("This app tells if a number is even or odd.",
        "AppInfo",
        MessageBoxButtons.OK,
        MessageBoxIcon.Information);
}
```

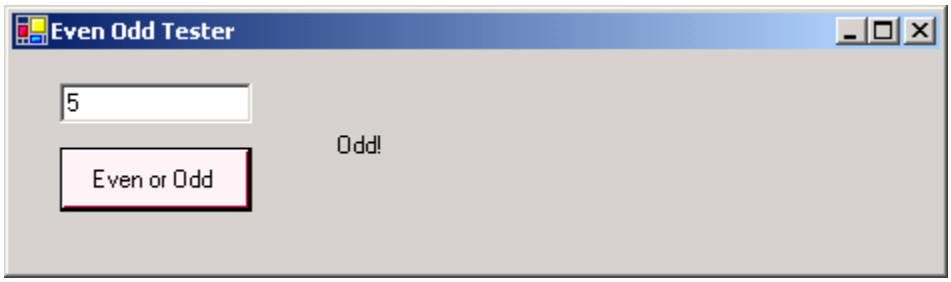
This code will set the background color of the button to LavenderBlush upon startup. It will also display a message box with the information "This app tells if a number is even or odd" with the title "AppInfo" and display an OK button and an "Information" icon:



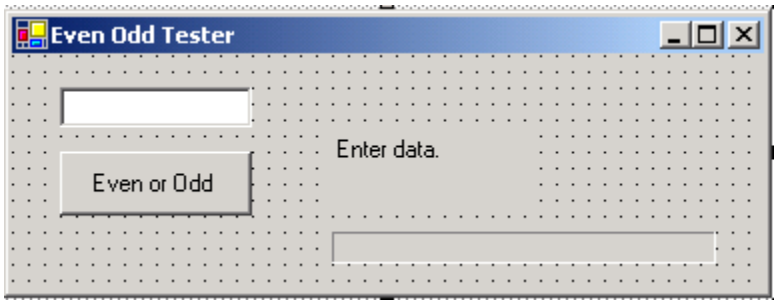
Now, let's add some code back to the buttonClicked event. We would like to take whatever data is entered in the textbox, convert it to an Integer, and then change the label's text to indicate if the integer is even or odd. The following code will do all of this. In this case we have added a try/catch block in the event of an error converting the integer (but ignore what kind of error may have occurred):

```
private void btnEvenOdd_Click(object sender, EventArgs e)
{
    int i;
    try
    {
        i = int.Parse(textBoxData.Text);
        if ((i % 2) == 0)
        {
            labelResult.Text = "Even!";
        }
        else
        {
            labelResult.Text = "Odd!";
        }
    }
    catch (Exception err)
    {
       labelResult.Text = "Error, invalid number.";
    }
}
```

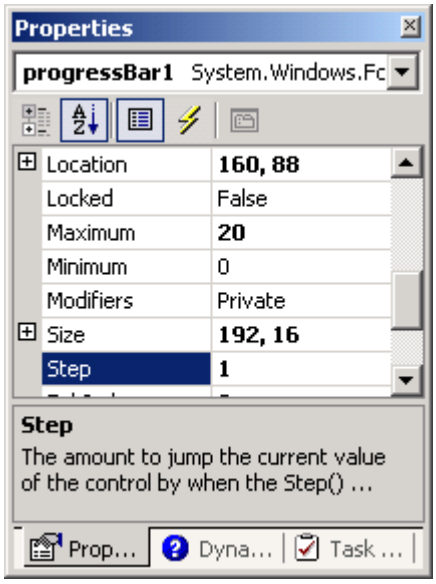
Here is a screen shot of an output case:



Before we leave this program, let's look at how to use the ProgressBar control. Add a ProgressBar control to the form, as shown below:



By default it will be given the name progressBar1. In its properties window, set the Minimum to 0 and the Maximum to 20 and the Step to 1:



The min and max specify the range of data values possible on the progress bar. Step indicates how much of the progress bar will be updated at a time. In this case, we create 20 progress bar intervals. If we set step to 2, we would only have 10 intervals. By setting the "Value" data member, we control how much of the progress bar is filled in.

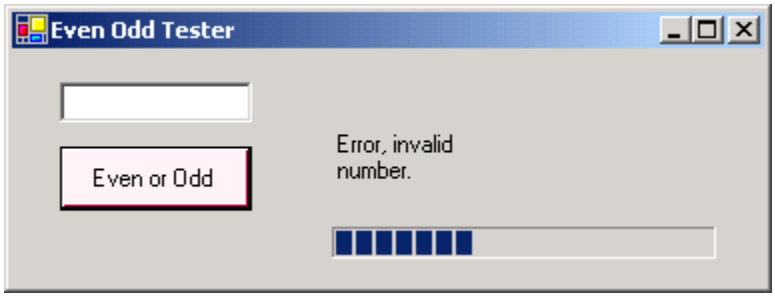
Normally, we would set the progress bar value during some long-running procedure so as to give feedback to the user that the program is operating and give them an idea how much longer to wait. In our program we have no such long-running procedure, so we will just pretend we have one. In our case, we will increment the value of the progress bar each time we click on the button.

In the buttonClicked event code, add the following at the bottom:

```
progressBar1.Value = progressBar1.Value + 1;
```

Every time we click the button, the progress bar's value will be incremented. The value starts at zero.

If we run the program and click the button a few times we should see the progress bar move forward:

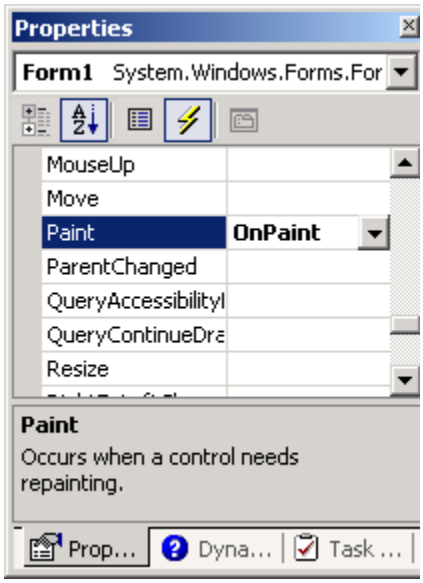


If we click the button more than 20 times we will have an error as we try to advance the progress bar's value beyond the maximum of 20. We could fix this in many ways; by allowing the progress bar to not be incremented past 20, or perhaps reset the progress bar to 0 if we hit 20:

```
if (progressBar1.Value == 20) progressBar1.Value = 0;
else progressBar1.Value = progressBar1.Value + 1;
```

Drawing Graphics

We can draw graphics in C# much like we draw graphics using Java. To play with some graphics, create a new Windows Application project. In the form's properties, go to the events and add an OnPaint method for the Paint event:



The OnPaint method will now be invoked whenever the form is redrawn. Recall that this is really used as a Delegate within the System's own Paint events and this method is passed to it so that the system knows what to invoke when a paint event occurs. By placing the drawing code in the Paint event it will always be updated correctly (e.g., if we placed the code somewhere else and the window was obscured, it may not be redrawn correctly when the obscuring item is moved out of the way).

We can now put code here that will draw whatever we like on top of the form. Just like in standard graphics format, the origin is the upper left corner. The x coordinates increase to the right, and the y coordinates increase down toward the bottom.

Here is some sample code we can add to the OnPaint method to draw various shapes on the screen:

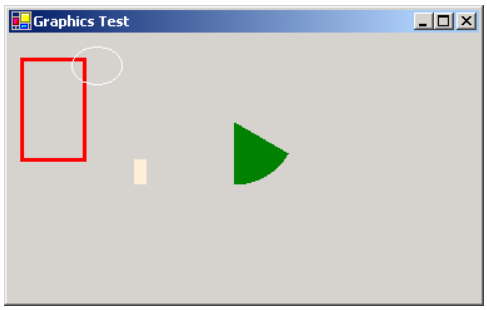
```
private void OnPaint(object sender, PaintEventArgs e)
{
      // Get the graphics object for the event
      Graphics g = e.Graphics;

      // New pen color
      Pen red = new Pen(Color.Red, 3);
      // Draw a rect at Width=50, Height=80 at coordinate 10,20
      g.DrawRectangle(red, 10, 20, 50, 80);

      // Ellipse drawn within bounding rectangle at 50,10
      g.DrawEllipse(new Pen(Color.Azure), 50, 10, 40, 30);

      // Brush = solid, hatched, texture...
      Brush bru = new SolidBrush(Color.PapayaWhip);
      // Fill in the rectangle
      g.FillRectangle(bru, 100, 100, 10, 20);

      // Draw part of a pie
      g.FillPie(new SolidBrush(Color.Green),
            130, 20, 100, 100, 30, 60);
}
```

First, we capture the Graphics object from the event arguments. The Graphics object is attached to the Form, so anything we draw on the graphics object will be displayed on the entire Form. We could attach code to the Paint event of specific controls to only paint within those controls, if we wish.

Next, we create a red pen and draw a rectangle using that pen. The rectangle takes the coordinates of the upper left corner then the width and height.

An ellipse is drawn in a similar fashion, by specifying the bounding rectangle that holds the ellipse.

Next we draw a solid rectangle using a Brush object. The Brush in this case is a solid color, but it is possible to create brushes that are hatched, texture, etc. Finally we draw part of a Pie slice using a green brush.

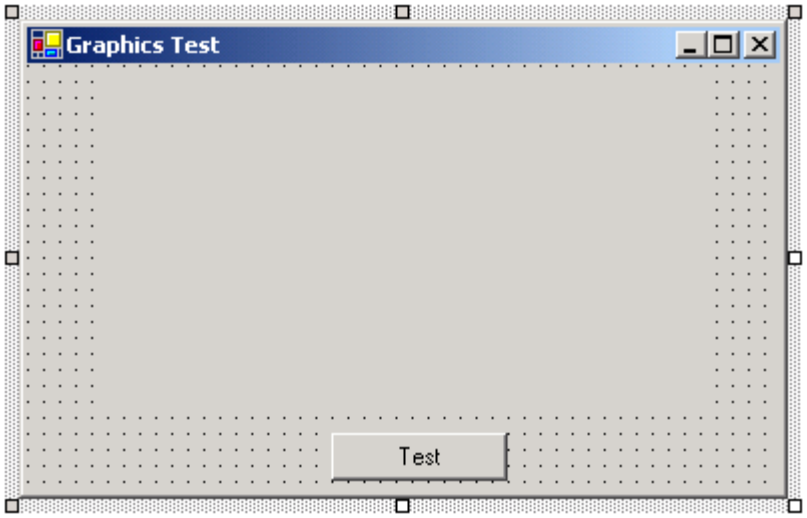


There are many other drawing tools available; see the text and online help for more details.

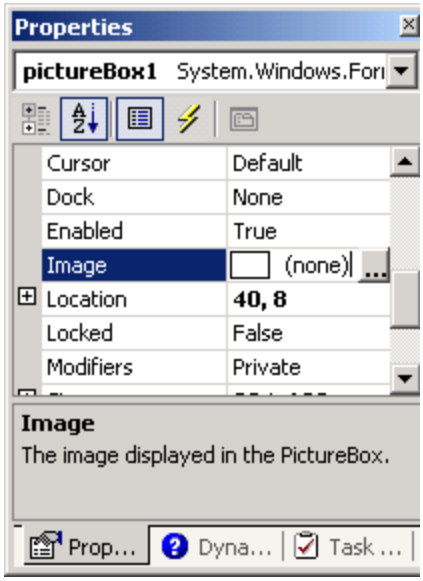
Working with Images

The homework assignment deals with images, so let's say a little bit about how to process and manipulate bitmap images.

First, create a new Windows Application project and add a PictureBox control and a Button to it. Set the text of the button to "Test". By default, the PictureBox object will be named pictureBox1 and the button will be named button1.

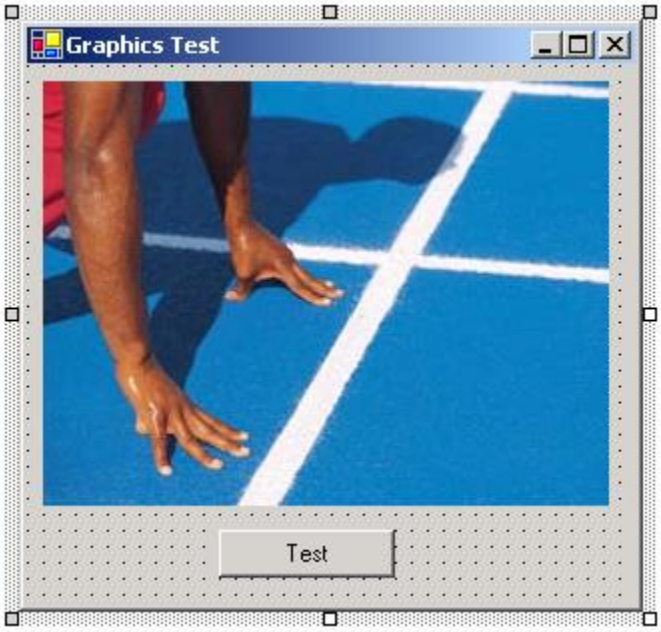


Now, select the pictureBox1 control and go to its properties. Find the image property in the list:



Click the "…" which will bring up a list of resources. Click on "import" and a file dialog to search for an image should be displayed. Find some image on your system. It should

then be displayed inside your pictureBox1 control. You may need to resize the control so that the entire image fits. In the picture below, I have selected the "sample.jpg" image that comes with Windows in the "My Documents/My Pictures" folder.



At this point you can run the application and it will display an image on the form. This is all you would need to do if you want to include static images in your application. The image data will be stored as part of the assembly of the executable, so you don't need to include the actual image file with your application.

Let's show how we can access individual colors of the image. Add the following code to the Button Click event of the "Test" button:

```
private void button1_Click(object sender, System.EventArgs e)
{
      // Get bitmap of the image
      Bitmap bmp = new Bitmap(pictureBox1.Image);
      int i;
      Color c;

      // Get color values for first line
      for (i=0; i < bmp.Width; i++)
      {
            c = bmp.GetPixel(i, 0);       // Get pixel at coordiate i,0
            Console.WriteLine("At coordinate " + i + ",0 " +
             " Color: Red=" + c.R + " Green=" + c.G + " Blue=" + c.B);
      }
}
```

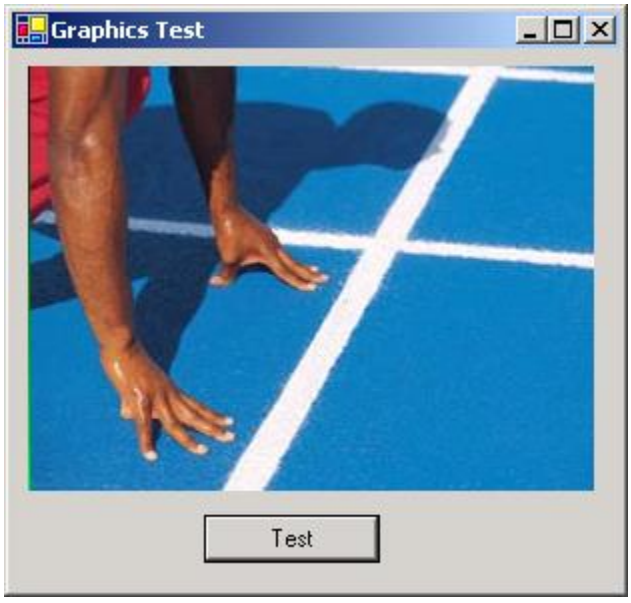
This code will output the Red, Green, and Blue values of each pixel on the first horizontal line of the image when we click the button. The output will show up in the Output window, if in debug mode. The output window is normally selected as one of the tabs in

the bottom right window pane (although it may be moved around, depending on your configuration of Visual Studio. If you're not sure where it is, you can find it from the V)iew menu).

We can also set the color of pixels if we like. Consider the following code:

```
private void button1_Click(object sender, System.EventArgs e)
{
	Bitmap bmp = new Bitmap(pictureBox1.Image);
	int i;
	Color c;

	// Set colors of first vertical line to a gradient of Green
	for (i=0; i < bmp.Height; i++)
	{
		// FromArgb takes Red, Green, Blue
		// Scale green from 0-255 depending on position in height
		c = Color.FromArgb(0,i*255/bmp.Height,0);
		bmp.SetPixel(0,i,c);
	}
	// For our changes to take effect, we must set the image property
	pictureBox1.Image = bmp;
}
```

This code will get a color of green and set it to each pixel in the first vertical line of the image. Note that we must reset the Image property to our bitmap at the end for the changes to take effect:



At this point we have covered enough to complete the homework (with the addition of other things such as arrays, control logic, and other data structures).

There is a lot more to cover regarding C# and particularly in developing Visual Studio .NET applications, but hopefully we have covered enough to give you an idea about the language and the development environment.

Intro to Windows Presentation Foundation, WPF

Windows Presentation Foundation, or WPF, has been billed as the successor to Windows Forms. It lets you develop applications in a way similar to Windows Forms if you wish, or to use a declarative style. The WPF model uses XAML, which stands for Extensible Application Markup Language, and is used with Windows 8, Windows Phone, Silverlight, and the Xbox, so if you learn it then you can develop on many different types of platforms.

Some advantages of using XAML and WPF over Windows Forms:

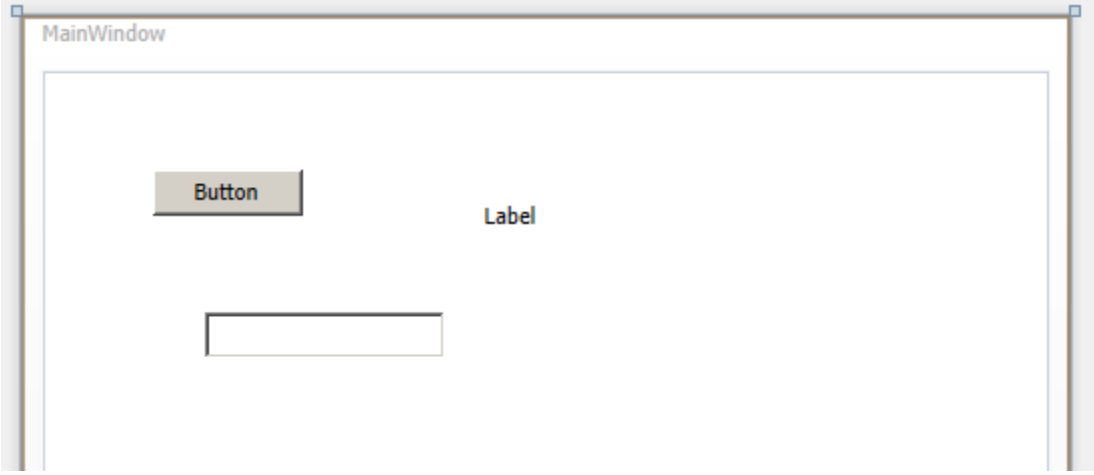
- Enforces separation of GUI display code from underlying logic code
- Ability to use graphical tools like Expression Blend to make fancy UI's
- Ability to use DirectX for better graphics performance than GDI+
- Built in support for 3D graphics
- Property Inheritance
- Styles
- Templates
- Broad platform support

The main disadvantage is the learning curve is a bit steep and somewhat non-traditional. If you have done modern HTML or XML then pieces will be more familiar to you.

Here we give only a brief introduction to some of the concepts in WPF.

To make a WPF Windows Application, select WPF Application when you make a new project:

We can add controls to the form in the designer, a lot like we did with Windows Forms. Here is a form with a button, textbox, and label.



By default these controls are placed inside a Grid. You can do a few handy things like lock an edge of a control for resizing. Here the button is locked to the left and right sides of the form:

When we resize the form, the button will resize with it.

If you look in the XAML window you can see the code that is generated to display the controls. In my case it looks like this:

```
<Window x:Class="WpfApplication2.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Button Content="Button" Height="23" Margin="12,73,12,0" Name="button1"
VerticalAlignment="Top" />
        <Label Content="Label" Height="28" Margin="202,29,241,0" Name="label1"
VerticalAlignment="Top" />
        <TextBox Height="23" HorizontalAlignment="Left" Margin="81,120,0,0"
Name="textBox1" VerticalAlignment="Top" Width="120" />
    </Grid>
</Window>
```

The same code in C# is quite a bit longer and would require creating a Button object, label object, setting their location, etc. (although this could be done for you by the Visual Studio Designer). You can write your code directly in the designer in XAML, which is often necessary to get the precise control you want, or use the designer.

If you like you can add events to the controls just like we did with Windows Forms. Here is an event for the button click:

```
private void button1_Click(object sender, RoutedEventArgs e)
{
    textBox1.Text = "You clicked the button";
}
```

You have even more flexibility designing the GUI if you use a tool like Microsoft's Expression Blend. It will create a project that you can open in Visual Studio and is intended to be used together (changes in one are reflected in the other). Expression Blend is good for designing the GUI but not so great for the coding part, while Visual Studio is great for coding but lacks some of the graphical tools in Expression Blend.

The real power of WPF comes by writing XAML code. Here we create a StackPanel of controls:

```
<StackPanel>
    <TextBlock Margin="20">Welcome to XAML</TextBlock>
    <Button Margin="10" HorizontalAlignment="Right">OK</Button>
    <Label>Hello there</Label>
</StackPanel>
```

We have lots of layout controls; the Stackpanel is one of the simpler ones. Here is a ScrollViewer which adds scroll bars to the items inside:

```
<ScrollViewer HorizontalScrollBarVisibility="Auto"
      VerticalScrollBarVisibility="Auto">
    <StackPanel>
        <TextBlock Margin="20">Welcome to XAML</TextBlock>
        <Button Margin="10" HorizontalAlignment="Right">OK</Button>
        <Label>Hello there</Label>
    </StackPanel>
</ScrollViewer>
```

The Canvas control lets us specify the distance between their left, top, right, bottom, and the canvas edge.

```
<Canvas>
    <Button Content="TopLeft" Width="85" Height="30"
            Canvas.Top="20" Canvas.Left="20"/>
    <Button Content="TopRight" Width="85" Height="30"
            Canvas.Top="20" Canvas.Right="20"/>
    <Button Content="Bottom" Width="85" Height="30"
            Canvas.Bottom="20" Canvas.Left="200"/>
</Canvas>
```

So we can control the layout using XAML somewhat like HTML; we also have some built in functions for animations and varying properties over time.

We can do this by adding triggers that run a storyboard. Here is an example to spin a button:

```
    <Window.Background>
        <RadialGradientBrush>
            <GradientStop Color="White" Offset="0"/>
            <GradientStop Color="Blue" Offset="1.2"/>
        </RadialGradientBrush>
    </Window.Background>
    <Window.Resources>
              <Style TargetType="Button">
                     <Setter Property="Width" Value="150"/>
                     <Setter Property="Height" Value="60"/>
                     <Setter Property="RenderTransform">
                        <Setter.Value>
                             <RotateTransform Angle="0" CenterX="75" CenterY="30"/>
                        </Setter.Value>
                     </Setter>
              </Style>
              <Storyboard x:Key="sbSpin">
                 <DoubleAnimation Duration="0:0:1" From="60" To="100"
                     Storyboard.TargetProperty="Height"/>
                 <DoubleAnimation Duration="0:0:1" From="0" To="360"
                      Storyboard.TargetProperty="RenderTransform.Angle"/>
              </Storyboard>
       </Window.Resources>
       <Grid>
              <Button Content="Spin Me!">
                     <Button.Triggers>
                            <EventTrigger RoutedEvent="Button.Click">
                                   <EventTrigger.Actions>
                                          <BeginStoryboard
Storyboard="{StaticResource sbSpin}"/>
                                   </EventTrigger.Actions>
                            </EventTrigger>
                     </Button.Triggers>
              </Button>
       </Grid>
```

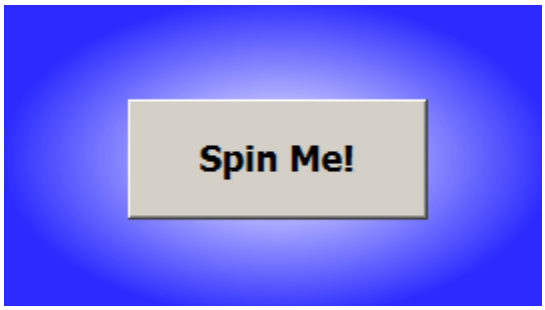
The background of the window uses a radial gradiant from white to blue.

The Window Resource lets tag lets us specify more detailed properties for resources. In this case we define a style for the button with accessibility to the width, height, and RenderTransform.Angle properties.

We define a Storyboard which lets us add elements that can run simultaneously or sequentially (to run sequentially add a BeginTime property). There are two DoubleAnimation elements, which is an animation of a value that is a double. The animation elapses over 0 hours, 0 minutes, and 1 second, and varies the height from 60 to 100. The angle varies similary from 0 to 360 degrees.

Finally, in the button, we add a Trigger when the button is clicked that begins (i.e. runs) the storyboard named sbSpin. Note the separation of action code from layout code.

The end result looks like this:



Imagine how you would write this using Windows Forms…

We also have a large degree of functionality to present 3D information. Here is an example rendering two spheres:

```
<Window
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
       x:Class="SingleMeshSpheres.Window1"
       x:Name="Window"
       Title="SingleMeshSpheres"
    Height="300" Width="500"
    Loaded="Window_Loaded">
    <Window.Resources>
        <Color x:Key="Color1" A="1" R="0" G="0" B="139"/>       <!-- DarkBlue -->
        <Color x:Key="Color2" A="1" R="173" G="216" B="230"/>   <!-- LightBlue -->
        <Transform3DGroup x:Key="CameraTransform">
            <RotateTransform3D>
                <RotateTransform3D.Rotation>
                    <AxisAngleRotation3D Axis="1,0,0"
                     Angle="{Binding ElementName=sliX,Path=Value}"/>
                </RotateTransform3D.Rotation>
            </RotateTransform3D>
            <RotateTransform3D>
                <RotateTransform3D.Rotation>
                    <AxisAngleRotation3D Axis="0,1,0"
                     Angle="{Binding ElementName=sliY,Path=Value}"/>
                </RotateTransform3D.Rotation>
            </RotateTransform3D>
        </Transform3DGroup>
        <PerspectiveCamera x:Key="Camera"
                    Position="0, 0, 3"
                    LookDirection="0, 0, -3"
                    UpDirection="0, 1, 0"
                    FieldOfView="60"
            Transform="{DynamicResource CameraTransform}"/>
    </Window.Resources>
    <Window.Background>
        <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
            <GradientStop Color="AliceBlue" Offset="0"/>
            <GradientStop Color="Blue" Offset="1"/>
```

```
        </LinearGradientBrush>
    </Window.Background>
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="*"/>
            <RowDefinition Height="20"/>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*"/>
            <ColumnDefinition Width="20"/>
        </Grid.ColumnDefinitions>

        <Slider Name="sliX" Grid.Row="0" Grid.Column="1" Orientation="Vertical"
               Minimum="-90" Maximum="90" Value="0"
               TickFrequency="10" TickPlacement="None">
            <Slider.LayoutTransform>
                <ScaleTransform ScaleX="1" ScaleY="-1"/>
            </Slider.LayoutTransform>
        </Slider>

        <Slider Name="sliY" Grid.Row="1" Grid.Column="0" Orientation="Horizontal"
               Minimum="-180" Maximum="180" Value="0"
               TickFrequency="10" TickPlacement="None"/>

        <UniformGrid Rows="1" Columns="2">
            <Viewport3D Margin="4,4,4,4" Camera="{DynamicResource Camera}">
                <ModelVisual3D>
                    <ModelVisual3D.Content>
                        <Model3DGroup x:Name="Sphere00">
                            <!-- Lights -->
                            <AmbientLight Color="{StaticResource Color1}"/>
                            <DirectionalLight Color="{StaticResource Color2}"
Direction="-1,-1,-1"/>
                        </Model3DGroup>
                    </ModelVisual3D.Content>
                </ModelVisual3D>
            </Viewport3D>

            <Viewport3D Margin="4,4,4,4" Camera="{DynamicResource Camera}">
                <ModelVisual3D>
                    <ModelVisual3D.Content>
                        <Model3DGroup x:Name="Sphere01">
                            <!-- Lights -->
                            <AmbientLight Color="{StaticResource Color1}"/>
                            <DirectionalLight Color="{StaticResource Color2}"
Direction="-1,-1,-1"/>
                        </Model3DGroup>
                    </ModelVisual3D.Content>
                </ModelVisual3D>
            </Viewport3D>

        </UniformGrid>
    </Grid>
</Window>
```

We make a 3D viewport with lights and a camera location. There is some code-behind to create two spheres with a different number of points, but the XAML determines the presentation.