8

Panels, Mouse Events, Colors

Review and Preview

You've seen and learned how to use lots of controls in the Visual C# toolbox. In this class, we begin looking at a very fun part of Visual C# - adding graphics capabilities to our projects. A key control for such capabilities is the panel. We will look at this control in detail. We will also look at ways for Visual C# to recognize mouse events and have some fun with colors. You will build an electronic blackboard project.

Panel Control

The **Panel** control is another Visual C# 'container' control. It is nearly identical to the **GroupBox** control (seen in Class 7) in behavior. Controls are placed in a Panel control in the same manner they are placed in the GroupBox. Radio buttons in a panel work as an independent group. Yet, panel controls can also be used to display graphics (lines, rectangles, ellipses, polygons, text). In this class, we will look at these graphic capabilities of the panel control. The panel is selected from the toolbox. It appears as:

In Toolbox:

On Form (default properties):

Properties

The panel properties are:

Property	Description
Name	Name used to identify panel. Three letter prefix for panel names is **pnl**.
BackColor	Sets panel background color.
Left	Distance from left side of form to left side of panel (X in properties window, expand **Location** property).
Top	Distance from top side of form to top side of panel (Y in properties window, expand **Location** property).
Width	Width of the panel in pixels (expand **Size** property).
Height	Height of panel in pixels (expand **Size** property).
Enabled	Determines whether all controls within panel can respond to user events (in run mode).
Visible	Determines whether the panel (and attached controls) appears on the form (in run mode).

Like the form and group box objects, the panel is a **container** control, since it 'holds' other controls. Hence, many controls placed in a panel will share the **BackColor** property (notice the panel does not have a Text property). To change this, select the desired control (after it is placed on the group box) and change the background color. Also, note the panel is moved using the displayed 'handle' identical to the process for the group box in the previous class.

Typical Use of Panel Control

The usual design steps for using a panel control are:

- Set **Name** property.
- Place desired controls in panel control.
- Monitor events of controls in panel using usual techniques.

Graphics Using the Panel Control

As mentioned, the panel control looks much like a group box and its use is similar. Panels can be used in place of group box controls, if desired. A powerful feature of the panel control (a feature the group box does not have), however, is its support of graphics. We can use the control as a blank canvas for self-created works of art! There are many new concepts to learn to help us become computer artists. Let's look at those concepts now.

Graphics Methods

To do graphics (drawing) in Visual C#, we use the built-in **graphics methods**. A **method** is a procedure or function, similar to the event methods we have been using, that imparts some action to an object or control. Most controls have methods, not just the panel. With the panel, a graphics method can be used to draw something on it. Methods can only be used in run mode. The C# code to use a method is:

```
objectName.MethodName(Arguments);
```

where ObjectName is the object of interest, MethodName is the method being used, and there may be some arguments or parameters (information needed by the method to do its task). Notice this is another form of the dot notation we use to set control properties in code. In this class, we will look at graphics methods that can draw colored lines. As you progress in your programming skills, you are encouraged to study the many other graphics methods that can draw rectangles, ellipses, polygons and virtually any shape, in any color. To use the panel for drawing lines, we need to introduce another concept, that of a **graphics object**.

Graphics Objects

You need to tell Visual C# that you will be using graphics methods with the panel control. To do this, you convert the panel control to something called a **graphics object**. Graphics objects provide the "surface" for graphics methods. Creating a graphics object requires two simple steps. We first declare the object using the standard declaration statement. If we name our graphics object **myGraphics**, the form is:

```
Graphics myGraphics;
```

This declaration is placed in the **general declarations** area of the code window, along with our usual variable declarations. Once declared, the object is created using the **CreateGraphics** method:

```
myGraphics = controlName.CreateGraphics();
```

where **controlName** is the name of the control hosting the graphics object (in our work, the **Name** property of the panel control). We will create this object in the form **Load** event of our projects.

Once a graphics object is created, all graphics methods are applied to this newly formed object. Hence, to apply a graphics method named **GraphicsMethod** to the **myGraphics** object, use:

```
myGraphics.GraphicsMethod(Arguments);
```

where **Arguments** are any needed arguments, or information needed by the graphics method.

There are two important graphics methods we introduce now. First, after all of your hard work drawing in a graphics object, there are times you will want to erase or clear the object. This is done with the **Clear** method:

```
myGraphics.Clear(Color);
```

This statement will clear a graphics object (myGraphics) and fill it with the specified **Color**. We will look further at colors next. The usual color argument for clearing a graphics object is the background color of the host control (controlName), or:

```
myGraphics.Clear(controlName.BackColor);
```

Once you are done drawing to an object and need it no longer, it should be properly disposed to clear up system resources. To do this with our example graphics object, use the **Dispose** method:

```
myGraphics.Dispose();
```

This statement is usually placed in the form **FormClosing** event method.

Our drawing will require colors and objects called pens, so let's take a look at those concepts. Doesn't it make sense we need pens to do some drawing?

Colors

Colors play a big part in Visual C# applications. We have seen colors in designing some of our previous applications. At design time, we have selected background colors (**BackColor** property) and foreground colors (**ForeColor** property) for different controls. Such choices are made by selecting the desired property in the properties window. Once selected, a palette of customizable colors appears for you to choose from.

Most graphics methods and graphics objects use color. For example, the pen object we study next has a **Color** argument that specifies just what color it draws with. Unlike control color properties, these colors cannot be selected at design time. They must be defined in code. How do we do this? There are two approaches we will take: (1) use built-in colors and (2) create a color.

The colors built into Visual C# are specified by the **Color** structure. We have seen a few colors in some our examples and projects. A color is specified using:

```
Color.ColorName
```

where **ColorName** is a reserved color name. There are many, many color names (I counted 141). There are colors like **BlanchedAlmond**, **Linen**, **NavajoWhite,** **PeachPuff** and **SpringGreen**. You don't have to remember these names. Whenever you type the word **Color**, followed by a dot (.), in the code window, the Intellisense feature of Visual C# will pop up a list of color selections. Just choose from the list to complete the color specification. You will have to remember the difference between BlanchedAlmond and Linen though!

If for some reason, the selection provided by the **Color** structure does not fit your needs, there is a method that allows you to create over 16 million different colors. The method (**FromArgb**) works with the **Color** structure. The syntax to specify a color is:

```
Color.FromArgb(Red, Green, Blue)
```

where **Red**, **Green**, and **Blue** are integer measures of intensity of the corresponding primary colors. These measures can range from 0 (least intensity) to 255 (greatest intensity). For example, **`Color.FromArgb(255, 255, 0)`** will produce yellow. Sorry, but I can't tell you what values to use to create **PeachPuff**.

It is easy to specify colors for graphics methods using the **Color** structure. Any time you need a color, just use one of the built-in colors or the **FromArgb** method. These techniques to represent color are not limited to just providing colors for graphics methods. They can be used anywhere Visual C# requires a color; for example, **BackColor** and **ForeColor** properties can also be set (at run-time) using these techniques. For example, to change your form background color to **PeachPuff**, use:

```
this.BackColor = Color.PeachPuff;
```

You can also define variables that take on color values. It is a two step process. Say we want to define a variable named **myRed** to represent the color red. First, in the general declarations area, declare your variable to be of type **Color**:

```
Color myRed;
```

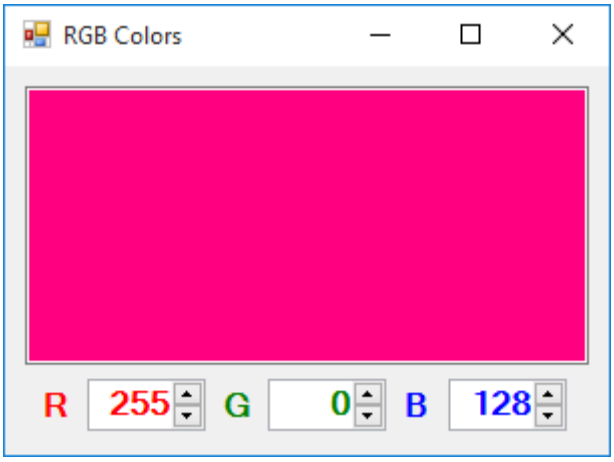
Then, define your color in code using:

```
myRed = Color.Red;
```

From this point on, you can use **myRed** anywhere the red color is desired.

Example

Go the course project folder (**\BeginVCS\BVCS Projects**) and open a project named **RGBColors**. Run this project. Three numeric updown controls are there: one to control the **Red** (R =) content, one to control the **Green** (G =) content, and one to control the **Blue** (B =) content. Set values for each and see the corresponding color assigned using the **FromArgb** function. Play with this project and look at all the colors available with this function. How long does it take to look at all 16 million combinations? A long time! The running project looks like this:



Stop the project when you're done playing with the colors. See if you can figure out how this little project works.

Pen Objects

As mentioned, many of the graphics methods (including the method to draw lines) require a **Pen** object. This virtual pen is just like the pen you use to write and draw. You can choose color and width. You can use pens built into Visual C# or create your own pen.

In many cases, the pen objects built into Visual C# are sufficient. These pens will draw a line **1** pixel wide in a color you choose (Intellisense will present the list to choose from). If the selected color is **ColorName** (one of the 141 built-in color names), the syntax to refer to such a pen is:

```
Pens.ColorName
```

Creating your own pen is similar to creating a graphics object, but here we create a **Pen** object. To create your own Pen object, you first declare the pen using:

```
Pen myPen;
```

This line goes in the **general declarations** area. The pen is then created using the **Pen constructor** function:

```
myPen = new Pen(Color, Width);
```

where **Color** is the color your new pen will draw in and **Width** is the integer width of the line (in pixels) drawn. The pen is usually created in the form **Load** method. This pen will draw a solid line. The **Color** argument can be one of the built-in colors or one generated with the **FromArgb** function.

Once created, you can change the color and width at any time using the **Color** and **Width** properties of the pen object. The syntax is:

```
myPen.Color = newColor;
myPen.Width = newWidth;
```

Here, **newColor** is a newly specified color and **newWidth** is a new integer pen width.

Like the graphics object, when done using a pen object, it should be disposed using the **Dispose** method:

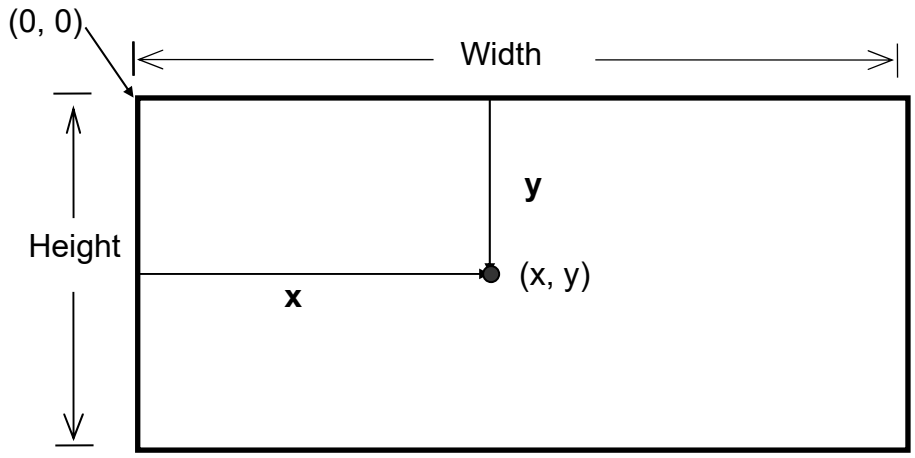
```
myPen.Dispose();
```

This disposal usually occurs in the form **FormClosing** method.

We're almost ready to draw lines – be patient! Just one more concept and we're on our way.

Graphics Coordinates

We will use Visual C# to draw lines using a method called the **DrawLine** method. Before looking at this method, let's look at how we specify the points used to draw and connect lines. All graphics methods use a default **coordinate system**. This means we have a specific way to refer to individual points in the control (a panel in our work) hosting the graphics object. The coordinate system used is:



We use two values (coordinates) to identify a single point in the panel. The **x** (horizontal) coordinate increases from left to right, starting at **0**. The **y** (vertical) coordinate increases from top to bottom, also starting at **0**. Points in the panel are referred to by the two coordinates enclosed in parentheses, or **(x, y)**. Notice how x and y, respectively, are similar to the Left and Top control properties. All values shown are in units of **pixels**.

At long last, we're ready to draw some lines.

DrawLine Method

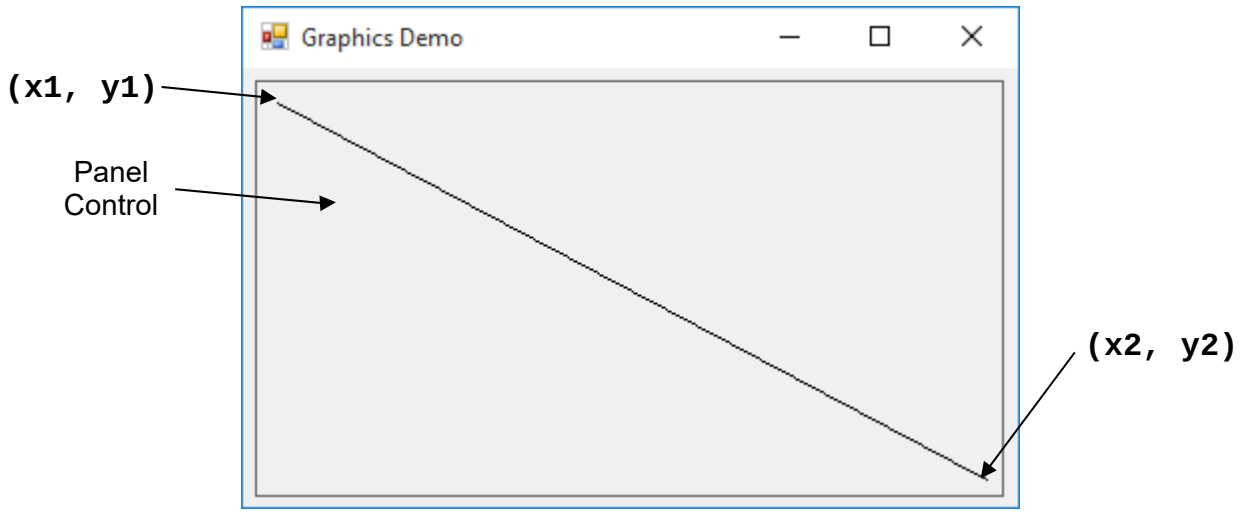
The Visual C# **DrawLine** method is used to connect two points with a straight-line segment. It operates on a previously created graphics object. If that object is **myGraphics** and we wish to connect the point (**x1**, **y1**) with (**x2**, **y2**) using a pen object **myPen**, the statement is:

```
myGraphics.DrawLine(myPen, x1, y1, x2, y2);
```

The pen object can be either one of the built-in pens or one you create using the pen constructor just discussed. Each coordinate value is an integer type. Using a built-in black pen (**Pens.Black**), the **DrawLine** method with these points is:

```
myGraphics.DrawLine(Pens.Black, x1, y1, x2, y2);
```

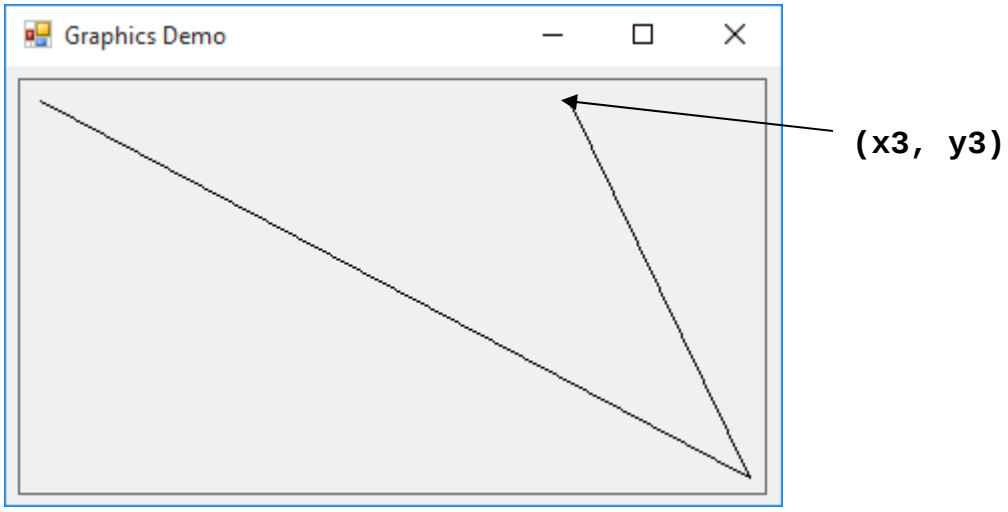
This produces on a panel (**MyGraphics** object):



To connect the last point (**x2, y2**) to another point (**x3, y3**), use:

```
myGraphics.DrawLine(Pens.Black, x2, y2, x3, y3);
```

This produces on a panel (**MyGraphics** object):



For every line segment you draw, you need a separate **DrawLine** statement. To connect one line segment with another, you need to save the last point drawn to in the first segment (use two integer variables, one for x and one for y). This saved point will become the starting point for the next line segment. You can choose to change the pen color at any time you wish. Using many line segments, with many different colors, you can draw virtually anything you want! We'll do that with the blackboard project in this class.

Graphics Review

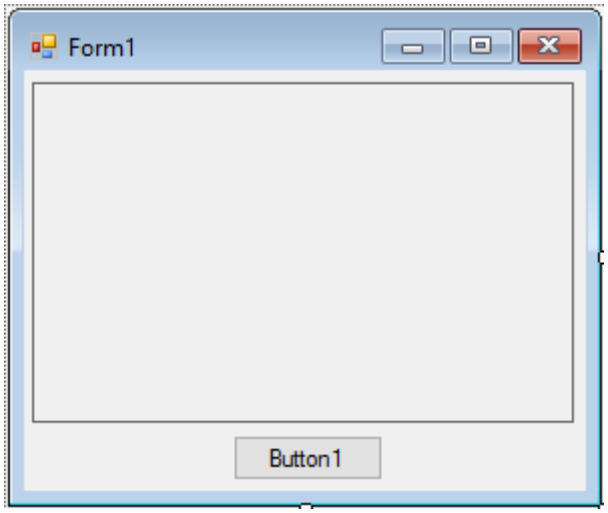
We've covered lots of new material here, so it's probably good to review the steps necessary to use the **DrawLine** method to draw line segments:

- **Declare** a **graphics object** in the general declarations area.
- **Create** a **graphics object** in the form Load event method.
- **Select** a **pen object** using the built-in Pens object or **create** your own **pen object**.
- **Draw** to **graphics object** using DrawLine method and specified coordinates.
- **Dispose** of **graphics object** and **pen object** (if created) in the form FormClosing event method.

The process is like drawing on paper. You get your paper (graphics object) and your pens. You do all your drawing and coloring and then put your supplies away!

Example

Start a new project in Visual C#. Place a panel control (name **panel1**) on the form. Make it fairly large. Set its **BackColor** property to white. Place a button control (name **button1**) on the form. My form looks like this:



Write down the **Width** and **Height** properties of your panel control (look at the **Size** property; my values are Width = 270 and Height = 170).

In the general declarations area of the code window, declare your graphics object using:

```
Graphics myGraphics;
```

In the **Form1_Load** event, add this line of code to create the graphics object:

```
myGraphics = panel1.CreateGraphics();
```

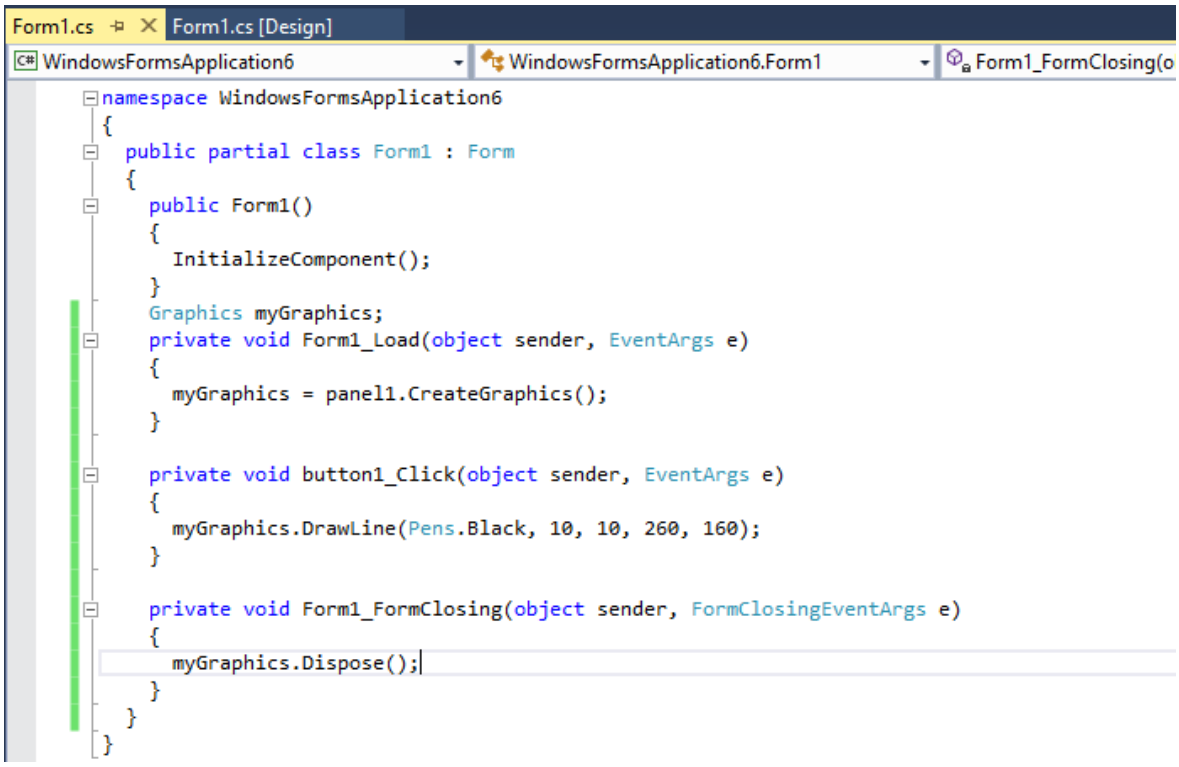
In the **button1_Click** event, write a line of code that draws a line starting at the point (10, 10) and goes to a point (Width – 10, Height – 10), where Width and Height are the dimensions of your panel control. Using a black pen, this line of code for my panel is:

```
myGraphics.DrawLine(Pens.Black, 10, 10, 260, 160);
```

And, in the **Form1_FormClosing** event, dispose of your graphics object using:

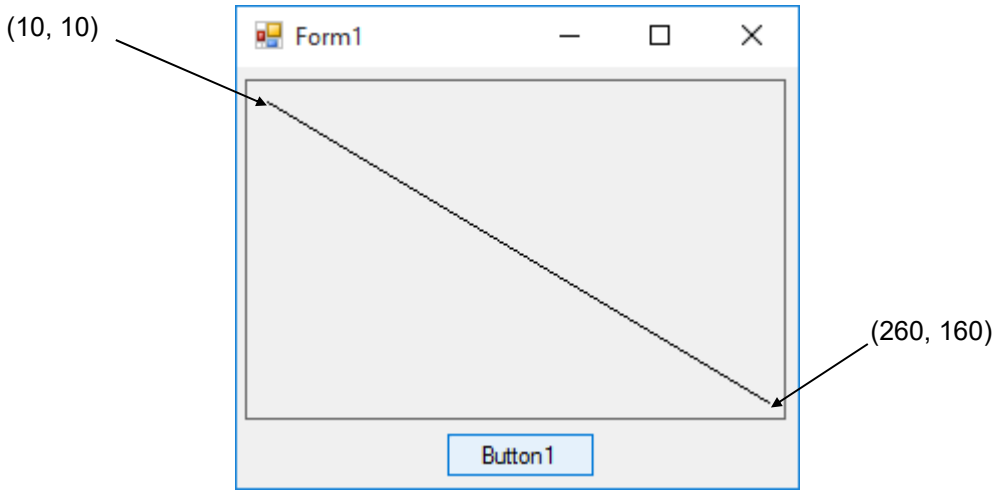
```
myGraphics.Dispose();
```

Make sure your code window looks like this:



Especially note the placement of the statement declaring the graphics object.

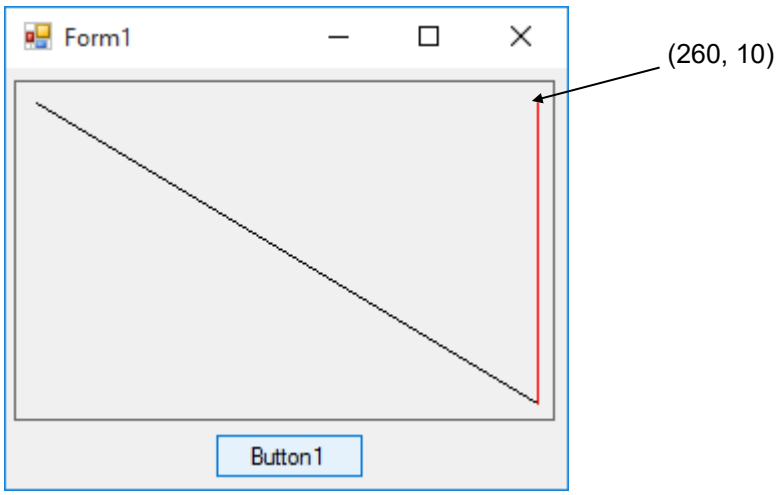
Run the project. Click the button. The code draws a black line from (10, 10), near the upper left corner, to (260, 160), near the lower right corner:



Stop the project and add this line after the current line in the **button1_Click** event:

```
myGraphics.DrawLine(Pens.Red, 260, 160, 260, 10);
```

Run the project again – click the button. A red line, connecting the last point to (260, 10) is added:



Stop the project. Let's create a wide pen. Add this line in the general declarations area to declare **myPen**:

```
Pen myPen;
```

Add this line of code in the **Form1_Load** event to create myPen as a blue pen with a drawing width of 10:

```
myPen = new Pen(Color.Blue, 10);
```

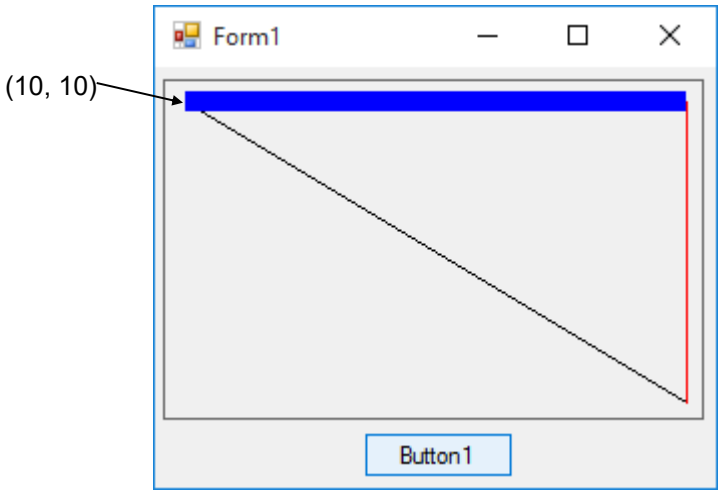
Add this line of code in the **Form1_FormClosing** event to dispose of myPen:

```
myPen.Dispose();
```

Now add this line after the lines already in the **button1_Click** event to draw a 'wide' blue line that completes a little triangle:

```
myGraphics.DrawLine(myPen, 260, 10, 10, 10);
```

Run the project, click the button and you should see:



Add more line segments, using other points and colors if you like. Try creating other pens with different colors and drawing widths. Save this project – we'll continue working with it. I think you get the idea of drawing. Just pick some points, pick some colors, and draw some lines. But, it's pretty boring to just specify points and see lines being drawn. It would be nice to have some user interaction, where points could be drawn using the mouse. And, that's just what we are going to do. We will use our newly gained knowledge about graphics methods to build a Visual C# drawing program. To do this, though, we need to know how to use the mouse in a project. We do that now.

C# - The Fifth Lesson

In the C# lesson for this class, we examine how to recognize mouse events (clicking and releasing buttons, moving the mouse) to help us build a drawing program with a panel control.

Mouse Events

Related to graphics methods are **mouse events**. The mouse is a primary interface for doing graphics in Visual C#. We've already used the mouse to **Click** on controls. Here, we see how to recognize other mouse events in controls. Many controls recognize mouse events - we are learning about them to allow drawing in panel controls.

MouseDown Event

The **MouseDown** event method is triggered whenever a mouse button is pressed while the mouse cursor is over a control. The form of this method is:

```
private void controlName_MouseDown(object sender,
MouseEventArgs e)
{
    [C# code for MouseDown event]
}
```

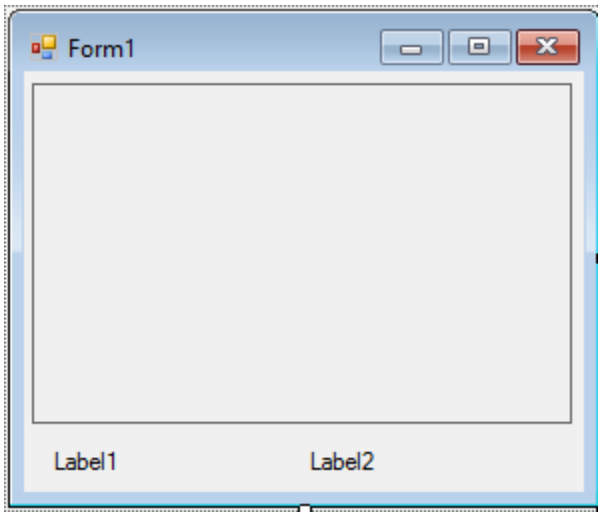
This is the first time we will use the arguments (information in parentheses) in an event method. This is information C# is supplying, for our use, when this event method is executed. Note this method has two arguments: **sender** and **e**. **sender** is the control that was clicked to cause this event (MouseDown) to occur. In our case, it will be the panel control. The argument **e** is an event handler revealing which button was clicked and the coordinate of the mouse cursor when a button was pressed. We are interested in three properties of the event handler **e**:

Value	Description
e.Button	Mouse button pressed. Possible values are: **MouseButtons.Left**, **MouseButtons.Center**, **MouseButtons.Right**
e.X	X coordinate of mouse cursor in control when mouse was clicked
e.Y	Y coordinate of mouse cursor in control when mouse was clicked

Only one button press can be detected by the MouseDown event - you can’t tell if someone pressed the left and right mouse buttons simultaneously. In drawing applications, the **MouseDown** event is used to initialize a drawing process. The point clicked is used to start drawing a line and the button clicked is often used to select line color.

Example

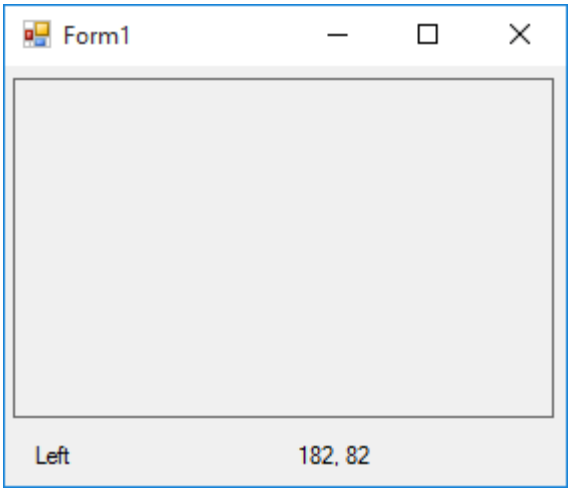
Let's try the MouseDown event with the example we just used with graphics methods. Recall we just have a panel control and a button on a form. Delete the button control from the form. Add two label boxes (one with default Name **label1** and one with default name **label2**) near the bottom of the form- we will use these to tell us which button was clicked and display the mouse click coordinate. My form looks like this:



Put these lines of code in the **panel1_MouseDown** event (select the event from the properties window for the panel control):

```
private void panel1_MouseDown(object sender, MouseEventArgs
e)
{
    switch (e.Button)
    {
        case MouseButtons.Left:
            label1.Text = "Left";
            break;
        case MouseButtons.Middle:
            label1.Text = "Middle";
            break;
        case MouseButtons.Right:
            label1.Text = "Right";
            break;
    }
    label2.Text = Convert.ToString(e.X) + "," +
Convert.ToString(e.Y);
}
```

Here, we use a switch structure to specify which button was clicked (displayed in label1 Text property) and we display e.X and e.Y (separated by a comma) in the label2 Text property. Run the project. Click the panel and notice the displayed button and coordinate. Here's an example of a point I clicked:



Try different mouse buttons. Click various spots in the panel and see how the coordinates change. Click near the upper left corner. Is (X, Y) close to (0, 0)? It should be. Play with this example until you are comfortable with how the MouseDown event works and what the coordinates mean. Stop and save the project.

MouseUp Event

The **MouseUp** event is the opposite of the MouseDown event. It is triggered whenever a previously pressed mouse button is released. The method format is:

```
private void controlName_MouseUp(object sender,
MouseEventArgs e)
{
        [C# code for MouseUp event]
}
```

Notice the arguments for MouseUp are identical to those for MouseDown. The only difference here is **e.Button** tells us which mouse button was released. In a drawing program, the **MouseUp** event signifies the halting of the current drawing process.

Example

Cut the code from the **panel1_MouseDown** method in our example (highlight the code, click the **Edit** menu, then **Cut**) and paste it in the **panel1_MouseUp** method (click **Edit**, then **Paste**). Make sure you select the correct method from the properties window before pasting. Run the project. Click the panel, move the mouse, then release the mouse. Note the displayed button and coordinates. Become comfortable with how the MouseUp event works and how it differs from the MouseDown event. Stop and save the project.

MouseMove Event

The **MouseMove** event is continuously triggered whenever the mouse is being moved. The event method format is:

```
private void controlName_MouseMove(object sender,
MouseEventArgs e)
{
        [C# code for MouseMove event]
}
```

And, yes, the arguments are the same. **e.Button** tells us which button is being pressed (if any) as the mouse is moving over the control and (e.X, e.Y) tell us the mouse position. In drawing processes, the **MouseMove** event is used to detect the continuation of a previously started line. If drawing is continuing, the current point is connected to the previous point using the current pen.

Example

Cut the code from the **panel1_MouseUp** method in our example and paste it in the **panel1_MouseMove** method. Run the project. Move the mouse over the panel. Notice the coordinates (X, Y) appear and continuously change as the mouse is moving. Click the panel and move the mouse. Notice the label boxes tell you which button was pressed and the current coordinates of the mouse in the panel. Stop the project.

You should now know how the three mouse events work and how they differ. Now let's use the panel control, DrawLine method, mouse events, pens and colors we've studied to build a fun drawing project

Project - Blackboard Fun

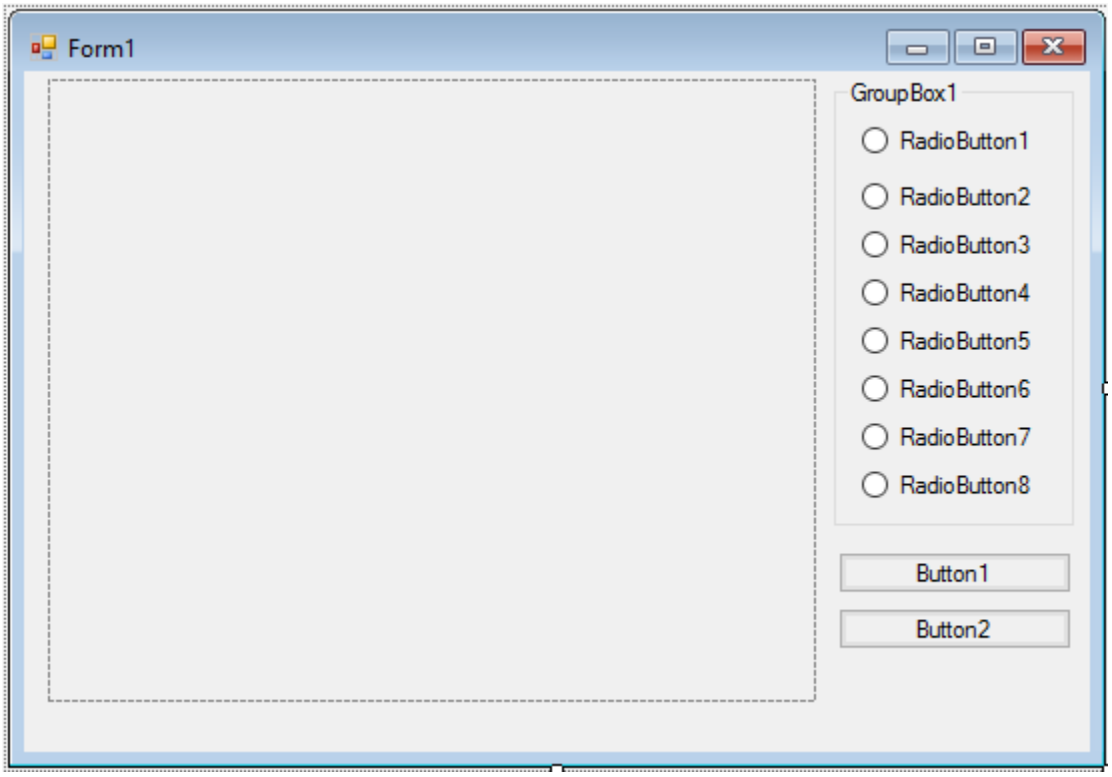
Have you ever drawn on a blackboard with colored chalk? You'll be doing that with the "electronic" blackboard you build in this project. This project is saved as **Blackboard** in the course projects folder (**\BeginVCS\BVCS Projects**).

Project Design

This is a simple project in concept. Using the mouse, you draw colored lines on a computer blackboard. A panel control will represent the blackboard. Radio buttons will be used to choose "chalk" color. Mouse events will control the drawing process. Two command buttons will be used: one to erase the blackboard and one to exit the program.

Place Controls on Form

Start a new project in Visual C#. Place a panel control (make it fairly large), a group box, and two command buttons on the form. Place eight radio buttons (used for color choice) in the group box. When done, my form looks like this:



Set Control Properties

Set the control properties using the properties window:

Form1 Form:

Property Name	Property Value
Text	Blackboard Fun
FormBorderStyle	Fixed Single
StartPosition	CenterScreen

panel1 Panel:

Property Name	Property Value
Name	pnlBlackboard
BorderStyle	Fixed3D
BackColor	Black (Of course! It's a Blackboard!)

groupBox1 Group Box:

Property Name	Property Value
Name	grpColor
Text	Color
BackColor	Black
ForeColor	White
Font Size	10
Font Style	Bold

radioButton1 Radio Button:

Property Name	Property Value
Name	rdoGray
Text	10 to 15 spaces (need some blank space to display color)

radioButton2 Radio Button:

Property Name	Property Value
Name	rdoBlue
Text	10 to 15 spaces

radioButton3 Radio Button:

Property Name	Property Value
Name	rdoGreen
Text	10 to 15 spaces

radioButton4 Radio Button:

Property Name	Property Value
Name	rdoCyan
Text	10 to 15 spaces

radioButton5 Radio Button:

Property Name	Property Value
Name	rdoRed
Text	10 to 15 spaces

radioButton6 Radio Button:

Property Name	Property Value
Name	rdoMagenta
Text	10 to 15 spaces

radioButton7 Radio Button:

Property Name	Property Value
Name	rdoYellow
Text	10 to 15 spaces

radioButton8 Radio Button:

Property Name	Property Value
Name	rdoWhite
Text	10 to 15 spaces

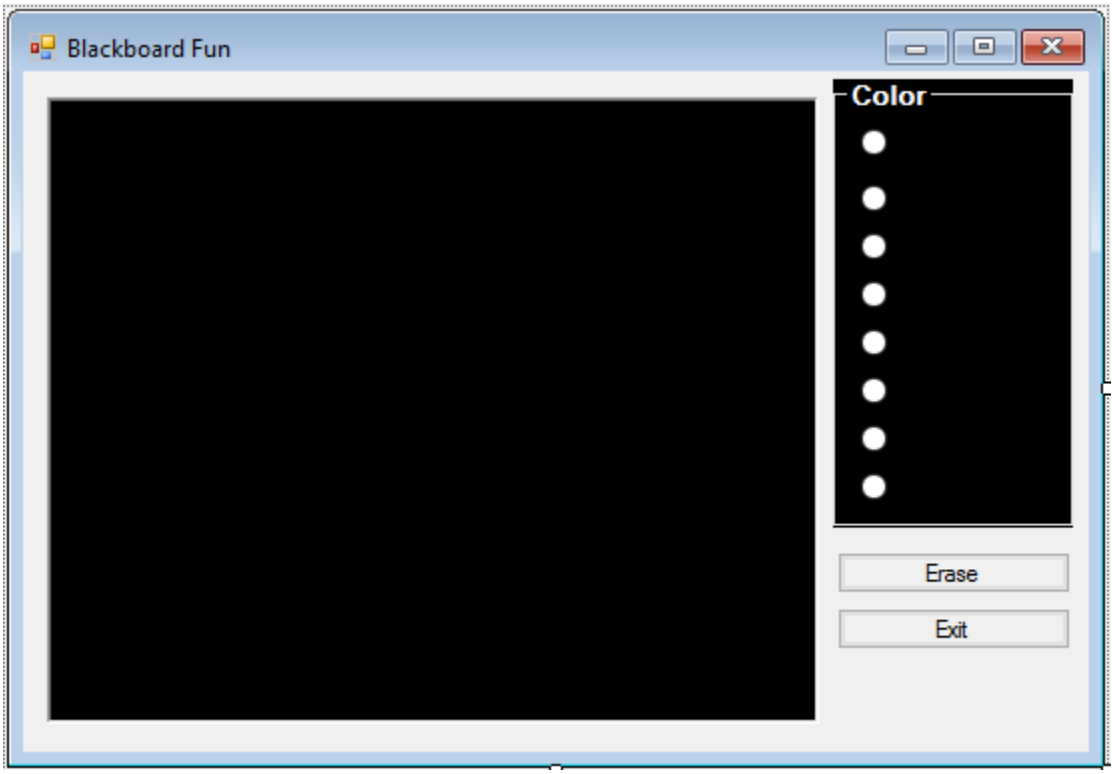
button1 Button:

Property Name	Property Value
Name	btnErase
Text	Erase

button2 Button:

Property Name	Property Value
Name	btnExit
Text	Exit

My form looked like this when I was done:



You may be asking – are you crazy? All the radio button Texts are empty spaces - how can these be used for picking colors? Wait a minute and you'll see.

Write Event Methods

This project will work like any paint type program you may have used. Click on a color in the **Color** group box (we'll see colors there soon) to choose a color to draw with. Then, move to the blackboard, left-click to start the drawing process. Drag the mouse to draw lines. Release the mouse button to stop drawing. It's that easy. Clicking **Erase** will clear the blackboard and clicking **Exit** will stop the program. Every step, but initializing a few things and stopping the program, is handled by the panel mouse events.

Three variables are used in this project. We need a Boolean variable (**mousePress**) that tells us whether the left mouse button is being held down. This lets us know if we should be drawing or not. We need two variables (**xLast** and **yLast**) that save the last point drawn in a line (we will always connect the "current" point to the "last" point). We also need a graphics object (**myGraphics**) and a pen object (**myPen**). Open the code window and declare these variables in the **general declarations** area:

```
bool mousePress;
int xLast;
int yLast;
Graphics myGraphics;
Pen myPen;
```

We need to establish some initial values. First, we create the graphics object (**myGraphics**) we will draw on and our pen object (**myPen**). We will set the pen to an initial drawing color of gray. How will we pick colors? Each radio button has a **BackColor** property. We set each BackColor property to its corresponding color using C# code. The user then sees each actual color and not some word describing it. The eight colors we will use are values from the Color structure. These colors were selected to look good on a black background. As seen, the

initial color will be Gray, so we set the **rdoGray** radio button **Checked** property to **true**. We also initialize **mousePress** to **false** (we aren't drawing yet). All this is done in the **Form1_Load** method:

```
private void Form1_Load(object sender, EventArgs e)
{
    // Create graphics and pen objects
    myGraphics = pnlBlackboard.CreateGraphics();
    myPen = new Pen(Color.Gray, 1);
    // Initialize the eight radio button colors
    rdoGray.BackColor = Color.Gray;
    rdoBlue.BackColor = Color.Blue;
    rdoGreen.BackColor = Color.LightGreen;
    rdoCyan.BackColor = Color.Cyan;
    rdoRed.BackColor = Color.Red;
    rdoMagenta.BackColor = Color.Magenta;
    rdoYellow.BackColor = Color.Yellow;
    rdoWhite.BackColor = Color.White;
    // Set initial color
    rdoGray.Checked = true;
    mousePress = false;
}
```

You'll see that this is pretty cool in how it works. This is a very common thing to do in Visual C# - initialize lots of properties in the form **Load** method instead of using the properties window at design time. It makes project modification much easier.

Each radio button needs a **CheckedChanged** event method to set the corresponding **myPen.Color** values. These eight one-line click events are:

```
private void rdoGray_CheckedChanged(object sender, EventArgs
e)
{
    // Gray
    myPen.Color = rdoGray.BackColor;
}

private void rdoBlue_CheckedChanged(object sender, EventArgs
e)
{
    // Blue
    myPen.Color = rdoBlue.BackColor;
}

private void rdoGreen_CheckedChanged(object sender, EventArgs
e)
{
    // Green
    myPen.Color = rdoGreen.BackColor;
}

private void rdoCyan_CheckedChanged(object sender, EventArgs
e)
{
    // Cyan
    myPen.Color = rdoCyan.BackColor;
}

private void rdoRed_CheckedChanged(object sender, EventArgs
e)
{
    // Red
    myPen.Color = rdoRed.BackColor;
}

private void rdoMagenta_CheckedChanged(object sender,
EventArgs e)
{
    // Magenta
    myPen.Color = rdoMagenta.BackColor;
}
```

```
private void rdoYellow_CheckedChanged(object sender,
EventArgs e)
{
    // Yellow
    myPen.Color = rdoYellow.BackColor;
}

private void rdoWhite_CheckedChanged(object sender, EventArgs
e)
{
    // White
    myPen.Color = rdoWhite.BackColor;
}
```

We'll code the two buttons before tackling the drawing process. The **btnErase** button simply clears the panel. The **btnErase_Click** method is:

```
private void btnErase_Click(object sender, EventArgs e)
{
    // Clear the blackboard
    myGraphics.Clear(pnlBlackboard.BackColor);
}
```

And, the **btnExit_Click** method is, as always:

```
private void btnExit_Click(object sender, EventArgs e)
{
    this.Close();
}
```

Now, let's code the drawing process. There are three events we look for:

- Left mouse button click - starts drawing
- Mouse moving with left mouse button pressed - continues drawing
- Left mouse button release - stops drawing

Each of these is a separate mouse event.

The **pnlBlackboard_MouseDown** event is executed when the left mouse button is clicked. When that happens, we set **mousePress** to **true** (we are drawing) and initialize the "last point" variables, **xLast** and **yLast**. That event method is:

```
private void pnlBlackboard_MouseDown(object sender,
MouseEventArgs e)
{
    // Start drawing if left click
    if (e.Button == MouseButtons.Left)
    {
        mousePress = true;
        xLast = e.X;
        yLast = e.Y;
    }
}
```

The **pnlBlackboard_MouseMove** event is executed when the left mouse button is being pressed (**mousePress** is **true**) and the mouse is moving over the panel. In this event, we connect the last point (**xLast, yLast**) to the current point (**e.X, e.Y**) using the **DrawLine** method with **myPen**. Once done drawing, the "last point" becomes the "current point." This code is:

```
private void pnlBlackboard_MouseMove(object sender,
MouseEventArgs e)
{
    // Draw a line if drawing
    if (mousePress)
    {
        myGraphics.DrawLine(myPen, xLast, yLast, e.X, e.Y);
        xLast = e.X;
        yLast = e.Y;
    }
}
```

The **pnlBlackboard_MouseUp** event is executed when the left mouse button is released. When that happens, we draw the last line segment and set **mousePress** to **false** (we are done drawing). That event method is:

```
private void pnlBlackboard_MouseUp(object sender,
MouseEventArgs e)
{
    // Finish line
    if (e.Button == MouseButtons.Left)
    {
        myGraphics.DrawLine(myPen, xLast, yLast, e.X, e.Y);
        mousePress = false;
    }
}
```

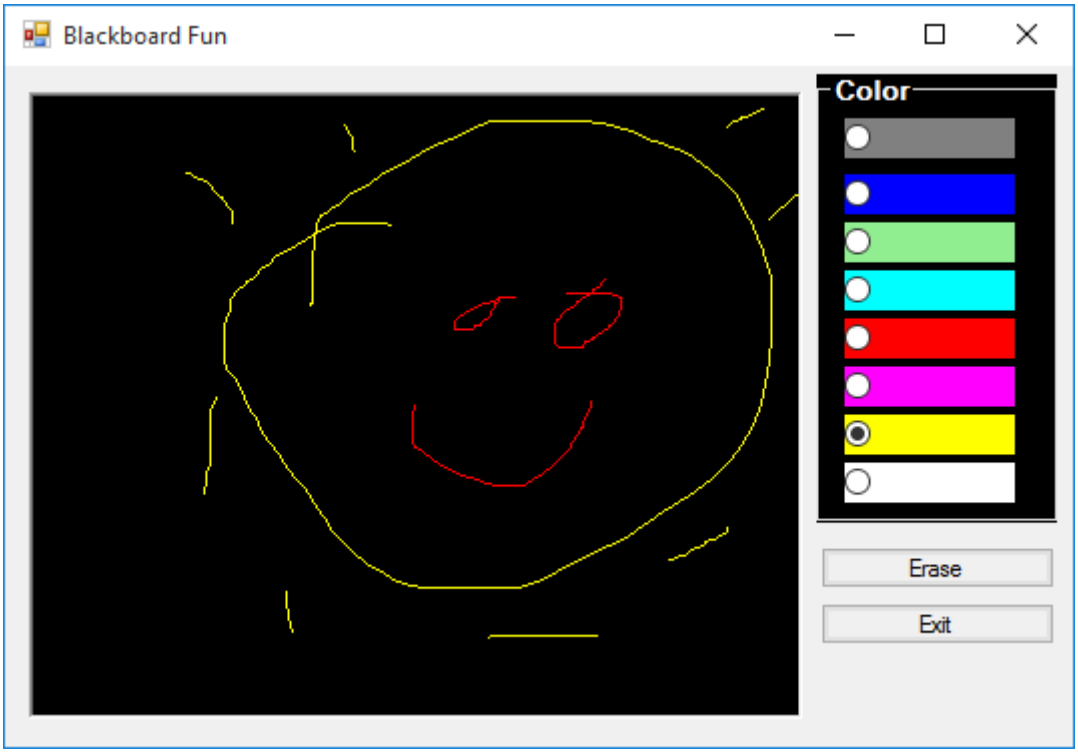
We're almost done. A last step is to dispose of our graphics and pen objects in the **Form1_FormClosing** event:

```
private void Form1_FormClosing(object sender,
FormClosingEventArgs e)
{
    myGraphics.Dispose();
    myPen.Dispose();
}
```

Save the project by clicking the **Save All** button in the toolbar.

Run the Project

Run the project. See how the radio button BackColor property is used to display colors? If the color choice areas are not very wide, make sure you set the Text property of each radio button to some blank space. Choose a color. Draw a line in the panel control. Try other colors. Draw something. Here's my attempt at art (a self-portrait):



I've had students draw perfect pictures of Fred Flintstone and Homer Simpson using this program. Make sure each color works. Make sure **Erase** works. Make sure **Exit** works. As always, thoroughly test your project. Save it if you had to make any changes while running it.

Do you see how simple the drawing part of this program is? Most of the code is used just to set and select colors. The actual drawing portion of the code (MouseDown, MouseMove, MouseUp events) is only a few lines of C#! This shows two things: (1) those drawing programs you use are really not that hard to build and (2) there is a lot of power in the Visual C# graphics methods.

Other Things to Try

The Blackboard Fun project offers lots of opportunity for improvement with added options. Have an option to set the pen **Width** property. This way you can draw with very skinny lines or very fat lines. Use a numeric updown control to set the value.

Add the ability to change the background color of the blackboard. Determine and build logic that allows drawing different colored lines depending on whether you press the left or right mouse button. For this, I'd suggest creating a left pen and a right pen. You will also need some way for the user to choose colors for each pen. Then, apply the appropriate pen in the various mouse events depending on what button is pressed.

See if you can figure out ways to get special effects. Here's one possibility to try. Delete (or 'comment out') these lines in the **pnlBlackboard_MouseMove** event:

```
xLast = e.X;
yLast = e.Y;
```

By doing this, the first point clicked (in the MouseDown event) is always the last point and all line drawing originates from this original point. Now, run the project again. Notice the "fanning" effect. Pretty, huh? Play around and see what other effects (change colors randomly, change pen width randomly). Have fun!

There is one effect of the Blackboard Fun project that is annoying. You may have discovered it. You may not have. In the upper right corner of the form is a small button with an "underscore" called the **minimize button**:

When you click this button, your application window disappears (is minimized) and is moved to the Windows task bar at the bottom of the screen. When you click your application name in the task bar, it will return to the screen. Go ahead and try it. Run the project and draw a few lines. Don't draw anything too elaborate – you'll soon find out why. Minimize your application, then restore your application by clicking the appropriate button in the Windows task bar. Where did your lines go?

Why did the lines disappear when the project went away for a bit? Visual C# graphics objects have no memory. They only display what has been last drawn on them. If you reduce your form to an icon on the task bar and restore it (as we just did), the graphics object cannot remember what was displayed previously – it will be cleared. Similarly, if you switch from an active Visual C# application to some other application, your Visual C# form may become partially or fully obscured. When you return to your Visual C# application, the obscured part of any graphics object will be erased. Again, there is no memory. Notice in both these cases, however, all controls are automatically restored to the form. Your application remembers these, fortunately! The controls are persistent. We also want **persistent graphics**.

The topic of persistent graphics is beyond the scope of this course. To eliminate this annoyance in the blackboard project, however, we will show you coding changes needed to add persistence. Only a few lines need to be changed. Make the changes if you like. In each case, the modified and/or new code is shown as shaded in gray. The idea is that we create a type of graphics object with memory (maintained in the **BackgroundImage** property of the panel control). The new **Form1_Load** method to do this is:

```
private void Form1_Load(object sender, EventArgs e)
{
    // Create graphics and pen objects
    pnlBlackboard.BackgroundImage = new
Bitmap(pnlBlackboard.Width, pnlBlackboard.Height,
System.Drawing.Imaging.PixelFormat.Format24bppRgb);
    myGraphics =
Graphics.FromImage(pnlBlackboard.BackgroundImage);
    myPen = new Pen(Color.Gray, 1);
    // Initialize the eight radio button colors
    rdoGray.BackColor = Color.Gray;
    rdoBlue.BackColor = Color.Blue;
    rdoGreen.BackColor = Color.LightGreen;
    rdoCyan.BackColor = Color.Cyan;
    rdoRed.BackColor = Color.Red;
    rdoMagenta.BackColor = Color.Magenta;
    rdoYellow.BackColor = Color.Yellow;
    rdoWhite.BackColor = Color.White;
    // Set initial color
    rdoGray.Checked = true;
    mousePress = false;
}
```

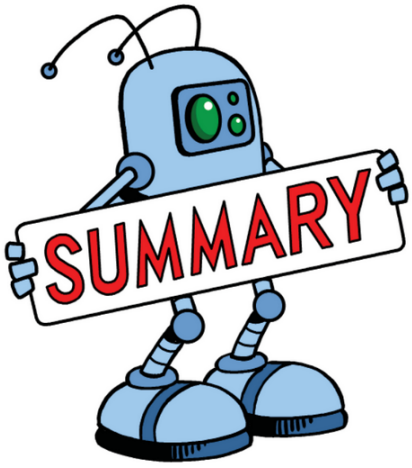
We also need to add a single line (shaded) to the **pnlBlackboard_MouseMove** event, **pnlBlackboard_MouseDown** event and **btnErase_Click** event. This line refreshes the added memory after each graphics method. The modified event methods are:

```
private void pnlBlackboard_MouseMove(object sender,
MouseEventArgs e)
{
    // Draw a line if drawing
    if (mousePress)
    {
        myGraphics.DrawLine(myPen, xLast, yLast, e.X, e.Y);
        pnlBlackboard.Refresh();
        xLast = e.X;
        yLast = e.Y;
    }
}

private void pnlBlackboard_MouseUp(object sender,
MouseEventArgs e)
{
    // Finish line
    if (e.Button == MouseButtons.Left)
    {
        myGraphics.DrawLine(myPen, xLast, yLast, e.X, e.Y);
        pnlBlackboard.Refresh();
        mousePress = false;
    }
}

private void btnErase_Click(object sender, EventArgs e)
{
    // Clear the blackboard
    myGraphics.Clear(pnlBlackboard.BackColor);
    pnlBlackboard.Refresh();
}
```

Try running the project again. You should now be able to minimize the project window without fear of losing your lovely work of art!



You've now had your first experience with graphics programming in Visual C# using the DrawLine method. You learned about the versatility of the panel control. You learned about three important control events to help in drawing: MouseDown, MouseMove, and MouseUp. And, you learned a lot about colors. In the next class, we'll continue looking at using graphics in projects. And, we'll look at some ways to design computer games.