Graphics

You can use graphics to enhance the user interface of your applications, generate graphical charts and reports, and edit or create images. The .NET Framework includes tools that allow you to draw lines, shapes, patterns, and text. This Lecture discusses how to create graphics and images using the classes in the *System.Drawing* namespace.

The System.Drawing Namespace

The .NET Framework includes the *System.Drawing* namespace, which enables you to create graphics from scratch or modify existing images. With the *System.Drawing* namespace, you can do the following:

- Add circles, lines, and other shapes to the user interface dynamically.
- Create charts from scratch.
- Edit and resize pictures.
- Change the compression ratios of pictures saved to disk.
- Crop or zoom pictures.
- Add copyright logos or text to pictures.

There are most important classes in the *System.Drawing* namespace, which you can use to build objects used for creating and editing images.

Of these classes, you use *Graphics* the most because it provides methods for drawing to the display device. The *Pen* class is used to draw lines and curves, while classes derived from the abstract class *Brush* are used to fill the interiors of shapes. In addition, you should be familiar with the *PictureBox* class (in the *System.Windows.Forms* namespace), which you can use in Windows Forms applications to display an image as part of the user interface.

Drawing Lines and Shapes

To draw on a form or control, follow these high-level steps:

1. Create a *Graphics* object by calling the *System.Windows.Forms.Control.CreateGraphics* method.
2. Create a *Pen* object.
3. Call a member of the *Graphics* class to draw on the form or control using the Pen.

Drawing begins with the *System.Drawing.Graphics* class. To create an instance of this class, you typically call a control's *CreateGraphics* method. Alternatively, as will be seen later, you can create a *Graphics* object based on an *Image* object if you want to be able to save the picture as a file . Once you create the *Graphics* object, you can call many methods to perform the drawing, including the following:

- ***Clear*** Clears the entire drawing surface and fills it with a specified color.
- ***DrawEllipse*** Draws an ellipse or circle defined by a bounding rectangle specified by a pair of coordinates, a height, and a width. The ellipse touches the edges of the bounding rectangle.
- ***DrawIcon*** **and** ***DrawIconUnstretched*** Draws the image represented by the specified icon at the specified coordinates, with or without scaling the icon.
- ***DrawImage,DrawImageUnscaled,*** **and** ***DrawImageUnscaledAndClipped*** Draws the specified *Image* object at the specified location, with or without scaling or cropping the image.
- ***DrawLine*** Draws a line connecting the two points specified by the coordinate pairs.
- ***DrawLines*** Draws a series of line segments that connect an array of *Point* structures.
- ***DrawPath*** Draws a series of connected lines and curves.

- ***DrawPie*** Draws a pie shape defined by an ellipse specified by a coordinate pair, a width, a height, and two radial lines. Note that the coordinates you supply with *DrawPie* specify the upper-left corner of an imaginary rectangle that would form the pie's boundaries; the coordinates do not specify the pie's center.
- ***DrawPolygon*** Draws a shape with three or more sides as defined by an array of *Point* structures.
- ***DrawRectangle*** Draws a rectangle or square specified by a coordinate pair, a width, and a height.
- ***DrawRectangles*** Draws a series of rectangles or squares specified by *Rectangle* structures.
- ***DrawString*** Draws the specified text string at the specified location with the specified *Brush* and *Font* objects.

To use any of these methods, you must provide an instance of the *Pen* class. Typically, you specify the color and width of the *Pen* class in pixels with the constructor. For example, the following code draws a red line 7 pixels wide from the upper-left corner (1, 1) to a point near the middle of the form (100, 100), as shown in Figure 6-1. To run this code, create a Windows Forms application and add the code to a method that runs in response to the form's *Paint* event:

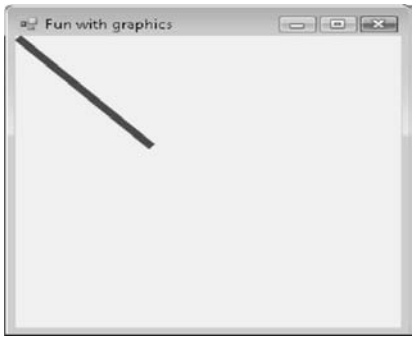


Figure 6-1 Use *Graphics.DrawLine* to create straight lines

```
Graphics g = this.CreateGraphics();
Pen p = new Pen(Color.Red, 5);
g.DrawLine(p, 1, 1, 100, 100);
```

Similarly, the following code draws a blue pie shape with a 60-degree angle, as shown in Figure 6-2:

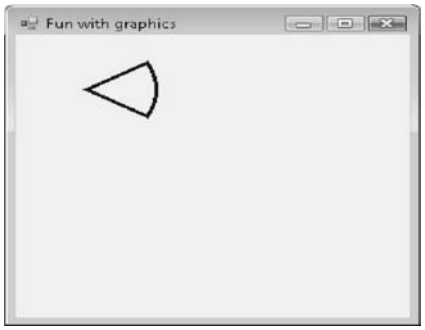


Figure 6-2 Use *Graphics.DrawPie* to create pie shapes

```
Graphics g = this.CreateGraphics();
Pen p = new Pen(Color.Blue , 2);
g.DrawPie(p, 100, 150,50, 50, 30, 350);
```

The *Graphics.DrawLines*, *Graphics.DrawPolygon*, and *Graphics.DrawRectangles* methods accept arrays as parameters to allow you to create more complex shapes. For example, the following code draws a purple, five-sided polygon, as shown in Figure 6-3:

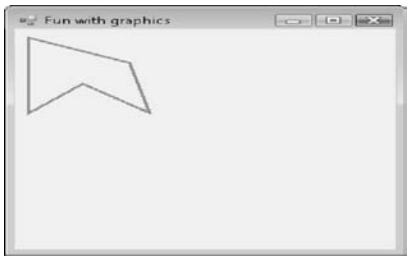


Figure 6-3 Use *Graphics.DrawPolygon* to create shapes made of multiple lines

```
Graphics g = this.CreateGraphics();
Pen p = new Pen(Color.Green , 3);
Point[] pointsArr =
            {
```

```
                new Point(50,30),
                new Point(120,45),
                new Point(140,65),
                new Point(100,100),
                new Point(80,125),
            };
g.DrawPolygon(p, pointsArr);
```

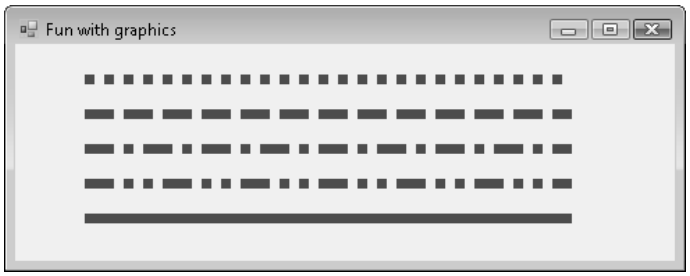
Note : Horizontal, Then Vertical:

When you pass coordinates to any .NET Framework method, you pass the horizontal (x) coordinate first, and then the vertical (y) coordinate second. In a 100-by-100 pixel image, (0,0) is the upper-left corner, (100,0) is the upper-right corner, (0, 100) is the lower-left corner, and (100,100) is the lower-right corner.

How to Customize Pens

Besides controlling the color and size of a pen, which are specified in the *Pen* constructor, you can also control the pattern and endcaps. The endcaps are the ends of the line, and you can use them to create arrows and other special effects.

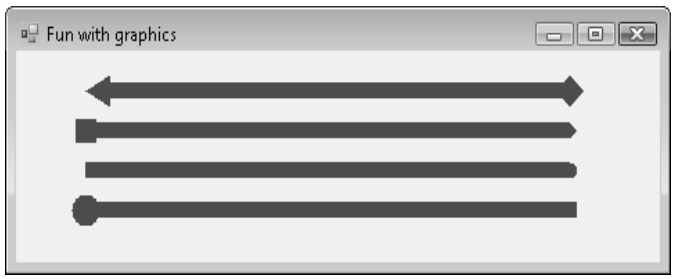
By default, pens draw solid lines. To draw a dotted line, create an instance of the *Pen* class, and then set the *Pen.DashStyle* property to one of these values: *DashStyle.Dash*, *DashStyle.DashDot*, *DashStyle.DashDotDot*, *DashStyle.Dot*, or *DashStyle.Solid*. The following code, which requires the *System.Drawing.Drawing2D* namespace, demonstrates each of these pen styles and creates the result shown in Figure 6-4:




```
Graphics g = this.CreateGraphics();
Pen p = new Pen(Color.Red, 7);
```

```
p.DashStyle = DashStyle.Dot;
g.DrawLine(p, 50, 25, 400, 25);

p.DashStyle = DashStyle.Dash;
g.DrawLine(p, 50, 50, 400, 50);

p.DashStyle = DashStyle.DashDot;
g.DrawLine(p, 50, 75, 400, 75);

p.DashStyle = DashStyle.DashDotDot;
g.DrawLine(p, 50, 100, 400, 100);

p.DashStyle = DashStyle.Solid;
g.DrawLine(p, 50, 125, 400, 125);
```

To control the endcaps and create arrows or callouts, modify the *Pen.StartCap* and *Pen.EndCap* properties using the *LineCap* enumeration. The following code demonstrates most of the pen cap styles and creates the result shown in Figure 6-5:




```
Graphics g = this.CreateGraphics();
Pen p = new Pen(Color.Red, 10);

p.StartCap = LineCap.ArrowAnchor;
p.EndCap = LineCap.DiamondAnchor;
g.DrawLine(p, 50, 35, 400, 35);

p.StartCap = LineCap.SquareAnchor;
p.EndCap = LineCap.Triangle;
g.DrawLine(p, 50, 60, 400, 60);
```

```
p.StartCap = LineCap.Flat;
p.EndCap = LineCap.Round;
g.DrawLine(p, 50, 85, 400, 85);

p.StartCap = LineCap.RoundAnchor;
p.EndCap = LineCap.Square;
g.DrawLine(p, 50, 110, 400, 110);
```

How to Fill Shapes

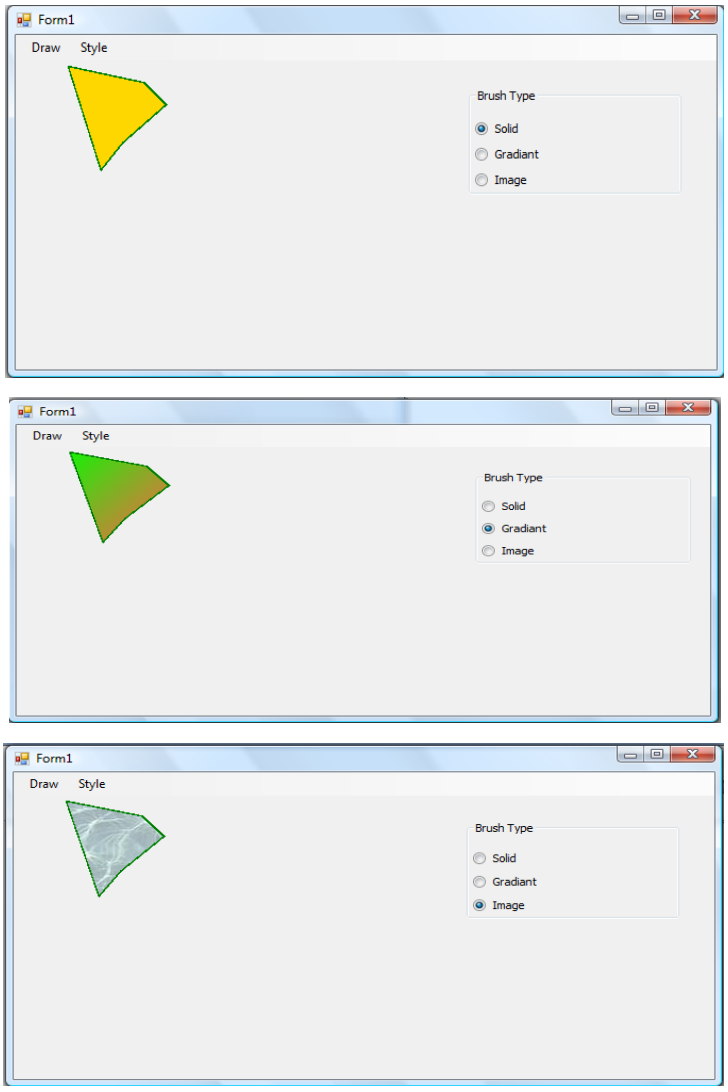
For most of the *Draw* methods, the *Graphics* class also has *Fill* methods that draw a shape and fill in the contents. These methods work exactly like the *Draw* methods, except they require an instance of the *Brush* class instead of the *Pen* class. The *Brus*h class is abstract, so you must instantiate one of the following child classes:

- *System.Drawing.Drawing2D.HatchBrush* Defines a rectangular brush with a hatch style, a foreground color, and a background color
- *System.Drawing.Drawing2D.LinearGradientBrush* Encapsulates a brush with a linear gradient that provides a visually appealing, professional-looking fill
- *System.Drawing.Drawing2D.PathGradientBrush* Provides similar functionality to *LinearGradientBrush*; however, you can define a complex fill pattern that fades between multiple points
- *System.Drawing.SolidBrush* Defines a brush of a single color
- *System.Drawing.TextureBrush* Defines a brush made from an image that can be tiled across a shape, like a wallpaper design

You can draw filled objects with an outline by first calling the *Graphics.Fill* method and then calling the *Graphics.Draw* method. You can use the same techniques to draw on controls, such as buttons or the instances of the *PictureBox* class. If you need to fill an entire *Graphics* object with a single color, call the *Graphics.Clear* method.

```
this.CreateGraphics().Clear(this.BackColor);
```

For example, the following code uses three types of filling (solid, linear gradiant , and texture bitmap) to fill a five-sided polygon, as shown in Figure :




```
Graphics g = this.CreateGraphics();
Brush b1 = new SolidBrush(Color.Gold );
Brush b2 = new LinearGradientBrush(new Point(30, 30),new Point( 130, 130),
                                                    Color.Lime, Color.Tomato );
Brush b3= new TextureBrush(Image.FromFile(@"D:\bmpS\images3.bmp"));

Point[] pointsArr =
            {
                new Point(50,30),
                new Point(120,45),
                new Point(140,65),
```

```
                new Point(100,100),
                new Point(80,125),
            };

if (radioButton1.Checked == true)
                g.FillPolygon(b1, pointsArr);
else
{
     if (radioButton2.Checked == true)
               g.FillPolygon(b2, pointsArr);
     else
               g.FillPolygon(b3, pointsArr)
}
```

Working with Images

The *System.Drawing.image* abstract class gives you the ability to create, load, modify, and save images such as .bmp files, .jpg files, and . tif files.

The *Image* class is abstract, but you can create instances of the class using the *Image.FromFile* method (which accepts a path to an image file as a parameter) and the *Image.FromStream* method (which accepts a *System.lO.Stream* object as a parameter). You can also use two classes that inherit *Image: System.Drawing.Bitmap* for still images, and *System.Drawing.Imaging.Metafile* for animated images.

Bitmap is the most commonly used class for working with new or existing images. The different constructors allow you to create a *Bitmap* from an existing *Image*, file, or stream, or to create a blank bitmap of a specified height and width. *Bitmap* contains two particularly useful methods that *Image* lacks:

- ***GetPixel*** Returns a *Color* object describing a particular pixel in the image. A pixel is a single colored dot in the image that consists of a red, green, and blue component.
- ***SetPixel*** Sets a pixel to a specified color.

However, more complex image editing requires you to create a *Graphics* object by calling *Graphics.Fromlmage.*

How to Display Pictures

To display in a form an image that is saved to the disk, load it with *Image.FromFile* and create a *PictureBox* control, and then use the *Image* to define *PictureBox.Image*. The following sample code (which requires a form with an instance of *PictureBox* named *pictureBoxl)* demonstrates this process. Change the filename to any valid image file:

```
Image img = Image.FromFile(@"D:\bmpS\images4.bmp");
pictureBox1.Image = bmp;
```

Similarly, the following code accomplishes the same thing using the *Bitmap* class:

```
Bitmap bmp = new Bitmap(@"D:\bmpS\images4.bmp");
pictureBox1.Image = bmp;
```

Alternatively, you can display an image as the background for a form or control by using the *Graphics.DrawImage* method. This method has 30 overloads, so you have a wide variety of options for how you specify the image location and size. The following code uses this method to set an image as the background for a form, no matter what the dimensions of the form are:

```
Bitmap bmp = new Bitmap(@"D:\bmpS\images3.bmp");
Graphics g= this.CreateGraphics();
g.DrawImage(bmp,50,50,200,200);
```

How to Create and Save Pictures

To create a new, blank picture, create an instance of the *Bitmap* class with one of the constructors that does not require an existing image. You can then edit it using the *Bitmap.SetPixel* method, or you can call *Graphics.FromImage* and edit the image using the *Graphics* drawing methods.

To save a picture, call *Bitmap.Save*. This method has several easy-to-understand overloads. Two of the overloads accept a parameter of type

System.Drawing.Imaging.Image-Format, for which you should provide one of the following properties to describe the file type: *Bmp*, *Emf*, *Exif*, *Gif*, *Icon*, *Jpeg*, *MemoryBmp*, *Png*, *Tiff*, or *Wmf*. *Jpeg* is the most common format for photographs, and *Gif* is the most common format for charts, screen shots, and drawings.

For example, the following code creates a blank 600-by-600 *Bitmap*, creates a *Graphics* object based on the *Bitmap*, uses the *Graphics.FillPolygon* and *Graphics.DrawPolygon* methods to draw a shape in the *Bitmap*, and then saves it to a file named Bm.jpg in the current directory. This code doesn't display any images and saves results to a picture file directly , and it requires the *System.Drawing.DrawinglD* and *System.Drawing.Imaging* namespaces:

```
Bitmap bmp = new Bitmap(600, 600);
for (int x = 0; x < bmp.Width; x++)
    for (int y = 0; y < bmp.Height; y++)
        bmp.SetPixel(x, y, Color.White);


Graphics gimage = Graphics.FromImage(bmp);

Pen p = new Pen(Color.RosyBrown , 10);
Point[] pointsArr =
{
    new Point(50,30),
    new Point(120,45),
    new Point(140,65),
    new Point(100,100),
    new Point(80,125),
};

gimage.DrawPolygon(p, pointsArr);
bmp.Save(@"D:\bmpS\images10.jpg", ImageFormat.Jpeg );
```