Basics of Application Development Using Windows Forms

Lesson 6

GDI+ capabilities

Contents

1. Introduction

So, we get to the important moment in learning the **.Net Framework** platform: **GDI+** library (Graphics Device Interface).

Although Web and Windows Forms applications allow building powerful applications that can display data from a variety of sources, sometimes this is not enough. For example, we may want to draw a text in a specific font and color, to display an image or other graphics. In order to display this kind of output, the application needs to instruct the operating system in part of how it should be displayed. **GDI+** just deals with these things.

1. What is GDI+?

GDI+ is a part of the Windows XP operating system; using it we can develop Windows and Web applications that allow you to work with vector and bitmap graphics, which will interact with graphical devices, such as a computer monitor, printer, or other display devices.

GUI (*Graphical User Interface*) applications, interacting with these devices, present data in a form understandable for the user, using the intermediary who retrieves the program data and sends them to the display device. The result is, for example, an image on the monitor. This intermediary is a **GDI+** library.

GDI+ does not interact directly with display devices, and uses a device driver for this purpose. Outputting a plain text, drawing lines or a rectangle, printing — all these are examples of **GDI+**.

Fig. 1.1 shows the interaction scheme described above.

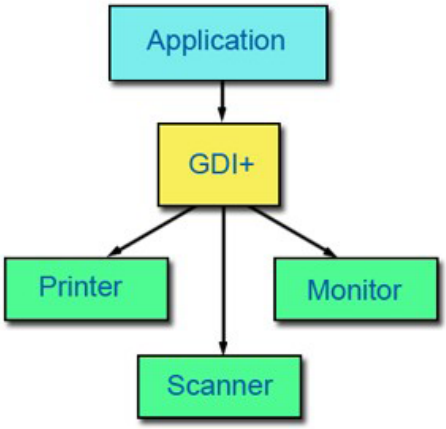

Fig. 1.1.

Now, let's consider the work of **GDI+** in more detail. Suppose our application draws a line. The line will be presented as a set of consecutive pixels, which has a starting and ending point. Drawing the line, the monitor needs to know in what place to draw it. How can we order the monitor to draw those pixels? For this purpose, in our application we use a **DrawLine()** method of the **GDI+** library. **GDI+** provides the operating system the instructions to display a line in the form of consecutive pixels.

2. Comparative analysis of GDI+ and GDI libraries

2.1. Analysis of the GDI operating principles

In order to get some drawing on the screen, we need such hardware as a video adapter. The computer gives specific commands for it, and it, in turn, causes the monitor to display what we need. In the market there is a variety of video adapters from different manufacturers, which have a specific set of commands and features. **GDI** allows us to abstract from these restrictions hiding the differences between these devices. **GDI** solves these problems on its own, there is no need for us to write a code for a particular driver. In order to output an image to a printer or monitor, we only need to call the appropriate methods.

Device Context (**DC**) is a Windows object that contains a set of graphic objects, information about their drawing attributes and determines the graphic modes of the display device.

Before displaying something, for example, on the monitor, the application needs to get the context before sending the output flow to the device. In the Net.Framework this is solved using a **System.Drawing.Graphics** class containing a device context.

Due to the **DC** interaction our application can be optimized, for example, when drawing a specific area on the screen. With the help of the device context we can determine in which coordinates and how to display what we need.

Graphic objects are represented by such classes as **Pen** — for drawing lines, **Brush** — for drawing and filling out forms, **Font** – for displaying a text, **Image**, and others.

2.2. Comparison of GDI and GDI+

GDI is a **Gdi32.dll** library; it was used in earlier versions of Windows and is based on the old Win32API with the C language functions. We can use this functionality in the . NET managed code. In order to apply the **GDI** library in our application, we need to import it via a **DLLImportAttribute** type. The following shows how to do it.

```
[System.Runtime.InteropServices.
                DllImportAttribute("gdi32.dll")]
```

After that, we will be able to use the functionality of the **gdi32.dll** library in our Net application. It is unlikely that we will use it in our applications, because in a managed language the **GDI+** library solves many problems.

GDI+ is a component of the WindowsXP and Windows Server2003 operating systems. This is a wrapper around the old **GDI** library, it is written in C++ and provides improved performance and more user-friendly programming model, appearing as **Gdiplus.dll** library. It can be used both in a Net managed (with the mounted Sysytem.**Drawing.dll** assembly) and unmanaged code.

In fact, the NET.Framework **GDI+** library also is a wrapper around the **GDI+** C++. It represents a more advanced API, including automatic memory management, cross-language integration, enhanced security, debugging, deployment, and more. Fig. 2.1 shows the relationship of the **GDI** and **GDI+** libraries.

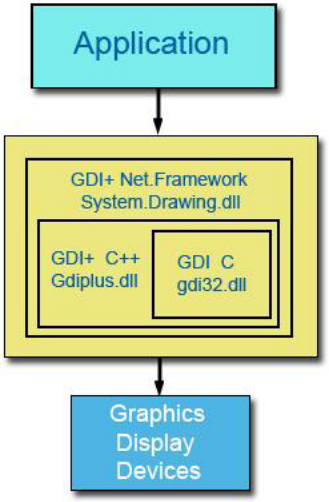

Fig. 2.1. The relationship of the GDI and GDI+ libraries.

3. System. Drawing Space

Basic functionality of the managed **GDI+** is represented by the NET.Framework library, defined in the **System.Drawing** namespace. In this space you can find classes that represent images, brushes, pens, fonts, and other types, allowing working with graphics. Additional functionality is provided by the subspaces: **System.Drawing.Design**, **System.Drawing.Drawing2D**, **System.Drawing.Imaging**, **System.Drawing.Printing**, **System.Drawing.Text**. The following table provides a brief description of these namespaces.

Table 3.1. Main GDI+ namespaces.

Namespace	Description
`System.Drawing`	Basic GDI+ namespace defines many types for basic rendering operations, for instance Graphics defines the methods and properties of drawing on the display devices, the Point and Rectangle types, for example, encapsulate the GDI+ primitives, the Pen class is used when drawing lines and curves, the classes derived from the Brush abstract type is used to fill internal areas of graphic shapes such as rectangles and ellipses. It is not supported in Windows and ASP.NET services.
`System.Drawing.Design`	The namespace contains the types that provide the core functionality for developing extensions of design-time

Namespace	Description
	user interface and placing them in the ToolBox, also includes predefined dialogs, for example: FontEditor is an editor for selection and configuration of a Font object, ColorEditor is an editor for visual picking a color, ToolBoxItem type is a base class designed for creating and visual displaying a ToolBox element on the toolbar. It is not supported in Windows and ASP.NET services.
`System.Drawing.Drawing2D`	The namespace is used to support two-dimensional and vector graphics. In turn, it is grouped by categories: a) types of brushes (PathGradientBrush and HatchBrush types allow filling geometric shapes with a pattern or gradient) b) enumerations related to drawing lines; LineCap and CustomLineCap types define cap styles for a line, LineJoin enumeration specifies how lines are joined together, PenAlignment enumeration specifies the alignment of the Pen object with respect to the virtual line, PenType enumeration specifies a line filling c) enumerations related to filling shapes and paths; HatchStyle enumeration specifies fill styles for a HatchBrush class, Blend specifies a blend for a LinearGradientBrush, FillMode enumerations specify the fill style for a GraphicsPath type

Namespace	Description
	d) geometric transformations; **Matrix** class represents a 3×3 matrix which contains information about transforming the vector graphics, image, or text, **MatrixOrder** enumeration specifies the order for transformation. It is not supported in Windows and ASP.NET services.
`System.Drawing.Printing`	It provides classes related to print-related services in Windows Forms, such as **PrintDocument**, **PrintSettings**, **PageSettings**; printing is performed via calling a **PrintDocument.Print()** method at the same time a PrintPage event is triggered, can be intercepted by the developer; **PrinterResolution** represents a resolution supported by a printer, a **PaperKind** enumeration defines standard paper sizes such as A4 or A3, and many other types. It is not supported in Windows and ASP.NET services, as well as in ASP.NET applications.
`System.Drawing.Imaging`	It contains classes that allow you to manipulate graphic images, for example, an **ImageFormat** class defines the image file format, a **ImageFlags** enumeration represents how the pixel data is contained in an image. It is not supported in Windows and ASP.NET services.
`System.Drawing.Text`	The space contains classes for managing fonts, for example, **InstalledFontCollection** provides a list of fonts

Namespace	Description
	installed in the system, **TextRendering-Hint** specifies the quality of text rendering, a **PrivateFontCollection** class provides access to the family of the client application fonts. It is not supported in Windows and ASP.NET services.

It was only a brief overview of the System.Drawing namespace, ahead there is a detailed introduction to the aspects of this technology.

4. Graphic primitives in GDI+

In **GDI+**, **Pen** is used to draw lines, curves and contours. It has several overloaded constructors allowing setting the color and the width of the brush. This object is used in drawing methods of the Graphics object. Below there is an example of using the **Pen** class.

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
    Pen pn = new Pen(Brushes.Blue, 5);
    pn.DashStyle = System.Drawing.Drawing2D.DashStyle.Dot;
    g.DrawEllipse(pn, 50, 100, 170, 40);
    g.Dispose();
}
```

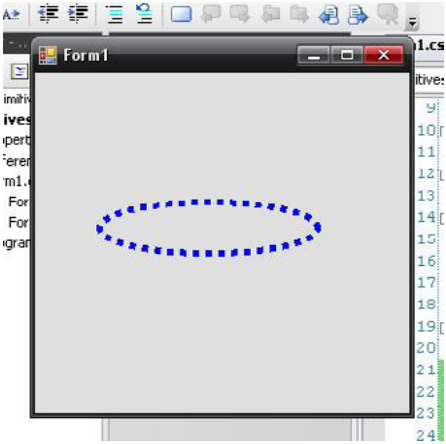

Fig. 4.1. Application of the Pen object

The above project is named **PenExample**.

In **GDI+**, **Brush** is used to fill the inner areas of the graphic shapes such as rectangles or ellipses.

Brush functionality is provided by the **System.Drawing** and **System.Drawing.Drawing2D** namespaces. For example, **Brush**, **SolidBrush**, **TextureBrush** and **Brushes** belong to the **System.Drawing** namespace, while **HatchBrush** and **GradientBrush** are set in the **System.Drawing.Drawing2D** namespace.

The **Brush** class is an abstract class, so it does not allow building instances of itself. It is a base for those types, such as **LinearGradientBrush**, **HatchBrush**, **GradientBrush** etc.

- **LinearGradientBrush** is used when you need to blend two colors and get a gradient fill.
- **HatchBrush** is used when a geometric shape should be filled with any pattern.
- **TextureBrush** allows filling geometric shapes with a bitmap image.

Below there is an example of using the brushes described above.

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
    Rectangle rect = new Rectangle(20, 20, 200, 40);
    LinearGradientBrush lgBrush =
                new LinearGradientBrush(
                rect, Color.Red, Color.Green, 0.0f, true);
    g.FillRectangle(lgBrush, rect);
    Rectangle rect2 = new Rectangle(20, 80, 200, 40);
    HatchBrush htchBrush = new HatchBrush(HatchStyle.
                Cross, Color.Blue);
```

```
    g.FillRectangle(htchBrush, rect2);
    TextureBrush txBrush = new TextureBrush(new
                Bitmap("Background.bmp"));
    Rectangle rect3 = new Rectangle(20, 140, 200, 40);
    g.FillRectangle(txBrush, rect3);
    g.Dispose();
}
```

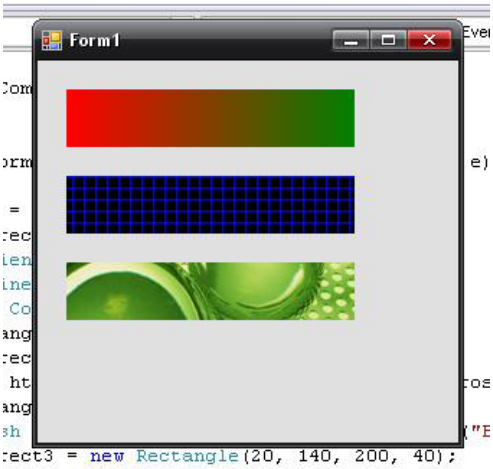

Fig. 4.2. Application of LinearGradientBrush, HatchBrush and TextureBrush

The above project is named **BrushesExample**.

Font is a font installed on the user's machine. **GDI** fonts are stored in the system directory **C:\WINDOWS\Fonts**. Fonts are divided into bitmap and vector. Bitmap fonts are rendered faster, but are difficult to transform, for example, to scale; in regard to memory consumption vector fonts are more voracious, but perfectly transformed.

In **GDI+**, **Font** overall functionality is provided by the **System.Drawing** namespace, additional functionality is provided by **System.Drawing.Text** space.

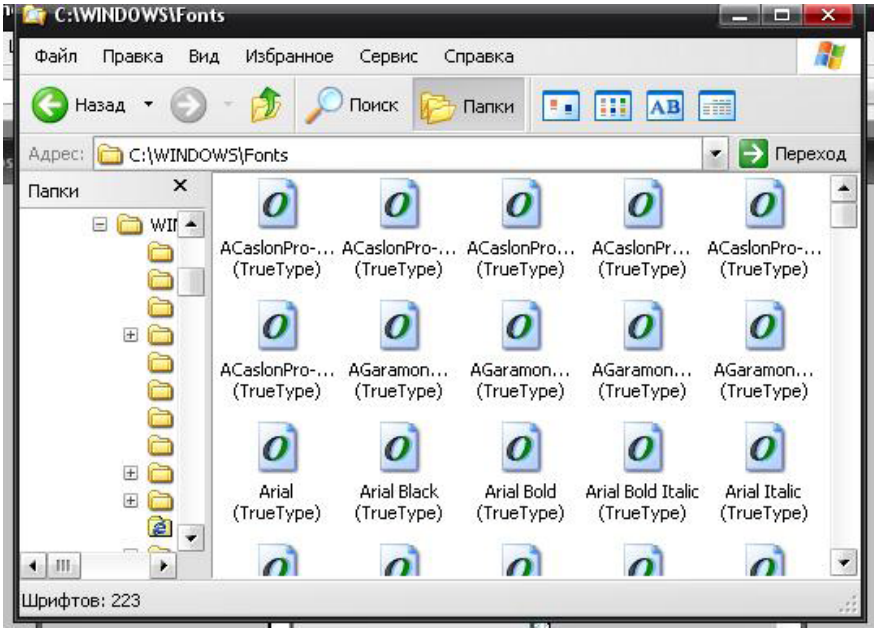

Fig. 4.3. The fonts available in Windows.

Font is used in the **Graphics.DrawString** method designed for drawing a text. The constructor of this class has a lot of overloads, and allows setting, for example, a typeface, size or font style; its instance can be created in various ways, for example:

```
Font f = new Font("Verdana", 12);
Font f = new Font("Verdana", 12, FontStyle.Bold);
Font f = new Font("Verdana", 12, FontStyle.Bold |
                                 FontStyle.Italic);
```

FontStyle enumeration sets a common style for a font.

```
public enum FontStyle
{
    Bold, Italic, Regular, Strikeout, Underline
}
```

Below you can find an example of the **Font** class usage:

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
    Font f = new Font("Verdana", 14, FontStyle.Bold |
                    FontStyle.Italic);
    g.DrawString("Hello Font!", f,Brushes.Blue, 30, 55);
    g.Dispose();
}
```

Fig. 4.4. Application of a Font object.

The above example is named **FontExample**.

Images, as well as fonts, are divided into two types: bitmap and vector. Bitmap ones are represented by a collection of one or more pixels, vector ones are represented by a collection of one or more vectors. Vector images may be transformed without loss of quality, while during the transformation of bitmap images the quality is lost.

In **GDI+**, images are represented by an abstract **Image** class, defined by the **System.Drawing** namespace. The image size is described by the properties such as **Widht**, **Height** and **Size**,

resolution – by the **HorizontalResolution** and **VerticalResolution** properties. Such methods as **FromImage()** and **FromStream()** allow creating an instance of an Image object from the specified file or stream. The **Image** object cannot be created directly, one should use other methods of creation, such as

```
Image im = new Bitmap("BgImage.bmp");
```

Let's consider an example of using the **Image** class.

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
    Rectangle rect = this.ClientRectangle;
    Image im = new Bitmap("BgImage.bmp");
    g.DrawImage(im, rect);
    g.Dispose();
}
```

Fig. 4.5. Application of the Image class.

The above project is named **ImageExample**.

Region is an entity describing the inner area of closed forms in **GDI+**, it is represented by the **System.Drawing.Region** class. With it, we can get the areas of intersection, exclusion and unification based on the mathematical set theory. For these purposes, the following methods are used:

- **Complement** performs the operation of union.
- **Exclude** performs the operation of exclusion.
- **Intersect** performs the operation of intersection.
- **Xor** performs the exclusive OR operation.

Below is an example of using a class region.

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.Clear(this.BackColor);

    //create two rectangles
    Rectangle rect1 = new Rectangle(40, 40, 140, 140);
    Rectangle rect2 = new Rectangle(100, 100, 140, 140);

    //create two regions
    Region rgn1 = new Region(rect1);
    Region rgn2 = new Region(rect2);
    g.DrawRectangle(Pens.Blue, rect1);
    g.DrawRectangle(Pens.Black, rect2);

    //define the area of intersection
    rgn1.Intersect(rgn2);

    //Fill it with red g.FillRegion(Brushes.Red, rgn1);
    g.Dispose();
}
```

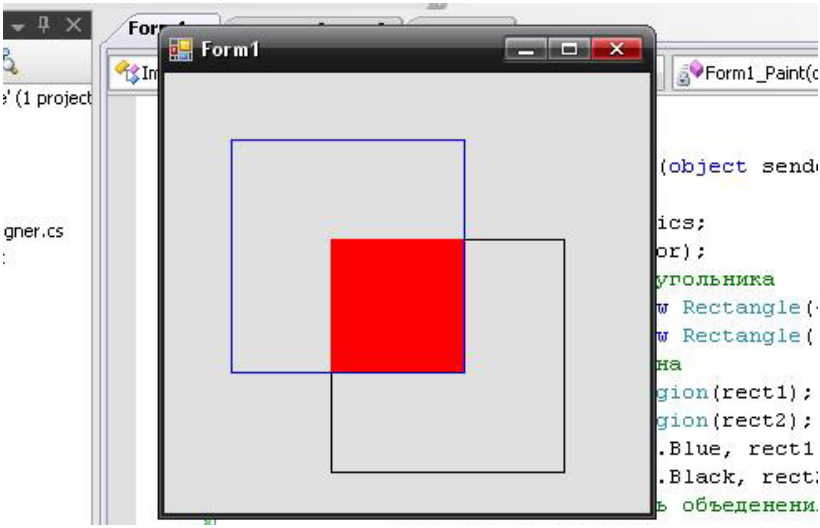

Fig. 4.6. Application of the Region class

The project is named **RegionExample**.

In **GDI+**, paths are represented by **GraphicsPath** class of the **System.Drawing.Drawing2D** namespace. The paths are series of closed lines, curves, including graphic objects such as rectangles, ellipses, and text. In applications, the paths are used, for instance, to draw contours, to fill interior regions, to create a clipping region.

GraphicsPath object can be formed by sequentially combining geometric shapes, using such methods as:

- **AddRectangle**,
- **AddEllipse**,
- **AddArc**,
- **AddPolygon**

In order draw a path, you should call the **DrawPath** method of the **Graphics** object.

If the task is to sequentially create multiple paths, it is necessary to call a **StartFigure()** method of a **GraphicsPath** object, then to form a path using the **AddXXX()** methods. By default, all paths are open; to explicitly close the path, you should call **CloseFigure()** method.

Below is an example of using the **GraphicsPath** class.

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;

    //create an array of points
    Point[] points = {
        new Point(5, 10),
        new Point(23, 130),
        new Point(130, 57)};

    GraphicsPath path = new GraphicsPath();
    //draw the first path
    path.StartFigure();
    path.AddEllipse(170, 170, 100, 50);

    //draw the first path
    g.FillPath(Brushes.Aqua, path);

    //draw the second path

    path.StartFigure();
    path.AddCurve(points, 0.5F);
    path.AddArc(100, 50, 100, 100, 0, 120);
    path.AddLine(50, 150, 50, 220);

    //close the path
    path.CloseFigure();
```

```
    //draw the forth path
    path.StartFigure();
    path.AddArc(180, 30, 60, 60, 0, -170);
    g.DrawPath(new Pen(Color.Blue, 3), path);
    g.Dispose();
}
```

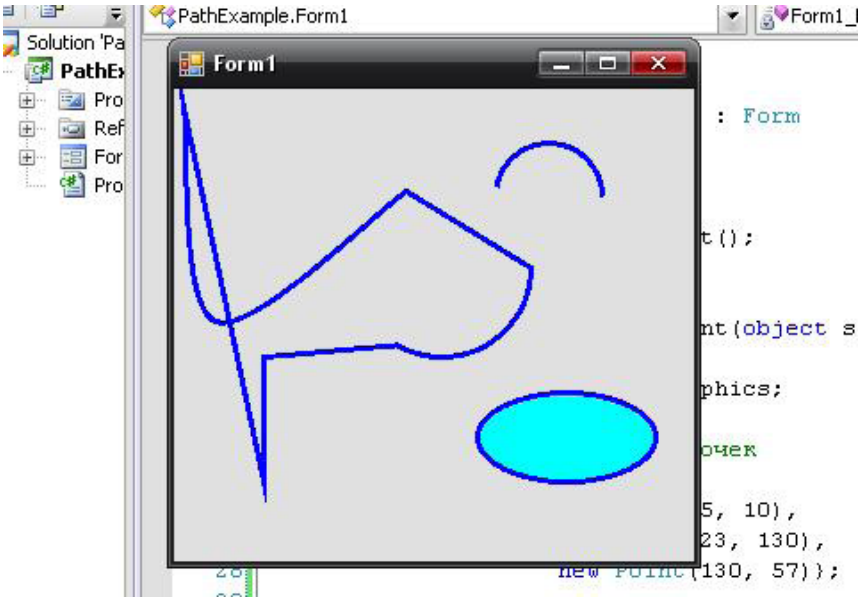

Fig. 4.6. Application of the GraphicsPath class.

The project is named **PathExample**.

5. Coordinate systems

In **GDI+**, there are three types of coordinate systems: **world** is a position of the point measured in pixels relative to the upper left corner of the document, **page** is a position of the point measured in pixels relative to the upper left corner of the client area, **device** are similar to page ones except for that the position of a point can be measured, for example, in millimeters or inches.

Before the graphic form is drawn on a surface using **GDI+**, it passes through transformation pipeline, first, world coordinates are converted to page ones (**world transformation**), then page coordinates are converted to device coordinates (**page transformation**) . Figure 5.1 shows a coordinate transformation pipeline.

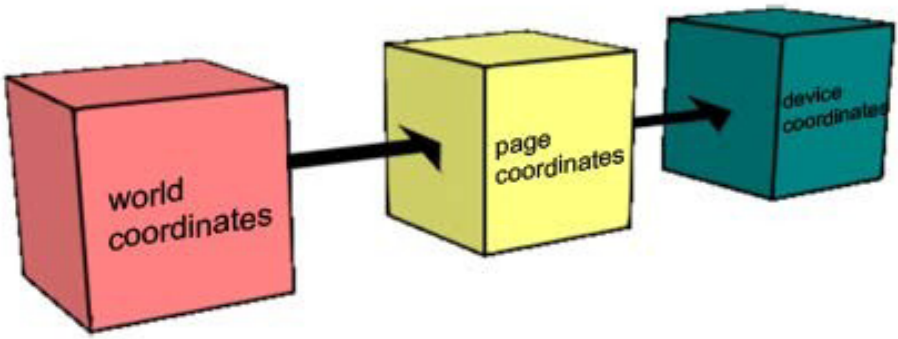

Fig. 5.1. The coordinate transformation pipeline.

Device coordinates shows how graphical objects will be displayed on output devices such as a monitor or printer. Default device coordinates are set in **Pixel**, and coincide with

the page ones. They can be changed using the **PageUnit** property of the **Graphics** class, e.g.:

```
Graphics g = e.Graphics;
g.PageUnit = GraphicsUnit.Inch;
```

The coordinates of graphical objects can be set by the **Point** structure or can be transferred in the drawing methods as parameters. **Point** is a structure, located in the **System.Drawing** space; it is an ordered pair of integers — **X** and **Y** coordinates that determine a point in the two-dimensional plane.

By default, the origin of the coordinate of all three systems is located at the point (0,0), which is located in the upper left corner of the client area and is given in pixels. We can change the origin of the page coordinate by the **TranslateTransform** method of the **Graphics** object. An example is shown below.

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
    //Move the origin of the page coordinate
    g.TranslateTransform(10, 50);
    Point A = new Point(0, 0);
    Point B = new Point(120, 120);
    g.DrawLine(new Pen(Brushes.Blue,3), A, B);

}
```

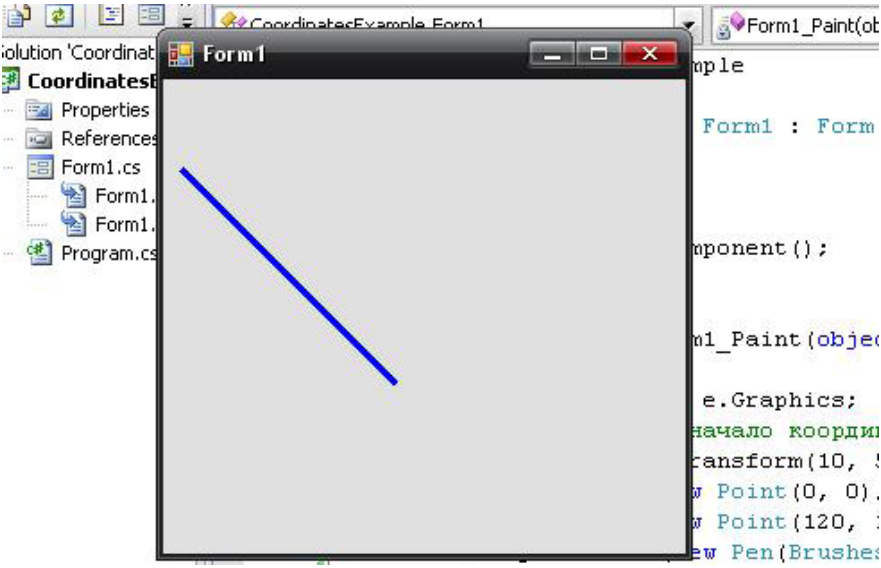

Fig. 5.2. Displacement of the origin of the page coordinates.

The project is called **CoordinatesExample**.

6. Graphics Class

6.1. Goals and objectives of the Graphics class

Graphics object is the heart of **GDI+**, in Net.Framework it is represented by the **System.Drawing.Graphics** class. It provides basic rendering functionality. As mentioned in the chapter 2, **Graphics** is associated with a specific device context. It contains methods and properties for drawing graphic objects, for example, the **DrawLine()** method draws a line, the **DrawPoligon()** draws a polygon, **DrawImage()** and **DrawIcon()** methods are used to draw a picture or icon.

6.2. Ways to gain access to an object of the Graphics class

Before you start drawing lines or a text using **GDI+**, we must receive an instance of the **Graphics** class. This object can be considered as the artist's drawing tool. To draw, we have to call the methods of this object.

Let's consider ways to retrieve the **Graphics** object using a simple example of the Windows Forms application, which will draw an inscription "Hello World!" on a form. Create a new project in Visual Studio, name it GraphicsExample.

Ways to retrieve the **Graphics** object:

1. Retrieving an instance of the **Graphics** object from the **PaintEventArgs** input parameter of the Paint event. In the form constructor, generate a **Paint** event handler.

```
public Form1()
{
    InitializeComponent();

    this.Paint += new PaintEventHandler(Form1_Paint);
}

private void Form1_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
    Font f = new Font("Verdana", 30, FontStyle.Italic);
    g.DrawString("Hello World!", f, Brushes.Blue, 10, 10);
}
```

The figure below shows the performance of our application.

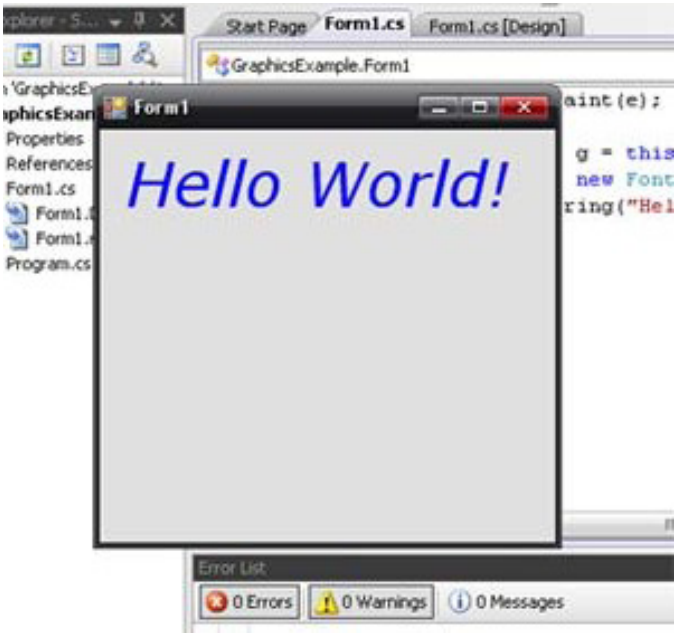

Fig. 6.1.

2. A similar result can be achieved as follows. Override the **OnPaint**() virtual method of the **Control** base class

```
protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e);
    Graphics g =e.Graphics;
    Font f = new Font("Verdana", 30, FontStyle.Italic);
    g.DrawString("Hello World!", f, Brushes.Blue, 10, 10);
}
```

3. The next way to get an instance of the **Graphics** object lies in calling the method of the **this.CreateGraphics()** form. Correct the code written above.

```
protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e);
    Graphics g = this.CreateGraphics();
    ...
}
```

4. The latter method, which we will consider, lies in getting **Graphics** usinf an object derived from the **Image** abstract class. Create a new project in the Visual Studio, call it GraphicsExample2. Drag a button on the form from the toolbar, call it **btnSave**, generate a method of the **Click** event handler for it. In this method we will execute image loading, then using GDI+ draw a text on it, frame it and saves with another name. The image file is to be created in a graphic editor, such as Photoshop, and put it in **bin>Debug** folder of the application.

```
private void btnSave_Click(object sender, EventArgs e)
{
    try
    {
        Bitmap myBitmap = new Bitmap(@"Background.bmp");

        //get the Graphics object
        Graphics gFromImage = Graphics.
            FromImage(myBitmap);
        Font f = new Font("Verdana", 8, FontStyle.Italic);
        string helloStr = "Hello World!";

        //meaure "Hello World!" using the method
        SizeF sz = gFromImage.MeasureString(helloStr, f);
        gFromImage.DrawString("Hello World!", f,
            Brushes.Blue, 10, 10);
        gFromImage.DrawRectangle(new Pen(Color.Red, 2),
            10.0F, 10.0F, sz.Width, sz.Height);

        //save the image on the disk
        myBitmap.Save(@"NewBackground.bmp");
        Rectangle regionRec = new Rectangle(new Point(0,0),
            myBitmap.Size);
        myBitmap.Dispose();
        gFromImage.Dispose();

        //This method performs redrawing of the client area
        this.Invalidate(regionRec);
    }
    catch { }
}
```

In case the **Background.bmp** file is absent, put our code in the try-catch block. The Graphics.**FromImage** method is responsible for creating an instance of the **Graphics** class, the **Invalidate** method is overloaded, it cannot have parameters

and is designed for force redrawing of the client area. Drawing area is defined with the **Rectangle** structure and pass it as a parameter to the **Invalidate()** method. Since the **Graphic** object uses a variety of unmanaged resources, it is recommended to use the **Dispose()** method.

Now, add a **Paint** event handler and fill it as shown below.

```
private void Form1_Paint(object sender, PaintEventArgs e)
{

    try
    {
        Bitmap myBitmap = new Bitmap(@"NewBackground.bmp");
        Graphics g = e.Graphics;
        g.DrawImage(myBitmap, 0, 0, 300, 200);
        myBitmap.Dispose();
        g.Dispose();
    }

    catch { }
}
```

Run the application, click **Save**. Go to Debug application folder, note that a new file called **NewBackground.bmp** was added.

6.3. General analysis of methods and properties of the Graphics class

Graphics class methods are divided into three categories: methods for filling inner areas of geometric shapes FillXXX(), drawing methods DrawXXX(), versatile methods, such as such as Clear(), Save(), MeasureString(), etc ..

Table 6.1. Basic Graphics drawing methods

Methods	Description

`DrawArc()`
`DrawBezier()`
`DrawBeziers()`
`DrawCurve()`
`DrawClosedCurve()`
`DrawIcon()`
`DrawEllipse()`
`DrawLine()`
`DrawLines()`
`DrawPie()`
`DrawPath()`
`DrawRectangle()`
`DrawString()`
`DrawImage()`
	The methods are used to draw graphical objects; almost all of the methods are overloaded and can accept a different number of parameters.

Table 6.2. Methods for filling Graphics inner areas

Methods	Description

`FillClosedCurve()`
`FillEllipse()`
`FillPath()`
`FillPie()`
`FillPoligon()`
`FillRectangle()`
`FillRegion()`
`FillRectangles()`
	Methods are used to fill the inner areas of graphic forms; almost all of the methods are overloaded and can accept a different number of parameters.

Table 6.3. Basic versatile Graphics methods

Methods	Description
`Clear()`	Leads to clearing of a **Graphics** object and fills it with a certain color, which is transmitted via the input parameter

Methods	Description
`ExcludeClip()`	Updates the clip region excluding the area specified by a Rectangle structure
`AddMetafileComment()`	Adds a comment to the current Metafile

`FromImage()`
`FromHdc()`
`FromHwnd()`
	These are methods allowing creating a Graphics object, such as an image, bitmap or GUI element

`GetNearestColor()`	Gets the nearest color specified with a Color structure
`IntersectClip()`	Updates the clip region of a Graphics object specified by the current clip region and Rectangle structure
`IsVisible()`	Returns true if a point is contained within the visible clip region
`MeasureString()`	Measures a string based on the specified font and returns SizeF
`MultiplyTransform()`	Multiplies the world transformation of a Graphics object and Matrix
`ResetClip()`	Resets the clip region of this Graphics to an infinite region
`ResetTransform()`	Resets the transformation matrix of a Graphics object
`Restore()`	Restores the state of a Graphics object defined by a GraphicsState input parameter
`RotateTransform()` `ScaleTransform()`	Applies rotation or scaling to the transformation matrix
`TransformPoints()`	Transforms an array of points from one coordinate system to another

Table 6.4. Graphics Key Properties

Methods	Description

`Clip`
`ClipBounds`
`VisibleClipBoud`
`IsClipEmpty`
`IsVisibleClipEmpty`
	There are properties allowing working with the clip region of a Graphics object.

`InterpolationMode`	Gets or sets the interpolation mode
`DpiX DpiY`	Returns the horizontal and vertical resolution of a Graphics object
`CompositionMode` `CompositionQuality`	There are properties allowing you to control the composition rendering quality
`Transform`	Gets or sets the world transformations specified by the Matrix type
`TextRenderingHint`	Gets or sets a rendering mode for a text
`TextContrast`	Gets or sets the amount of gamma correction for a text
`VisibleClipBounds`	Gets the bounding rectangle of the visible clipping region
`SmoothingMode`	Gets or sets the rendering quality for a Graphics object

Most of the drawing methods are overloaded and can accept a different number of parameters. For example, the **DrawLine** method has the following overloads:

- public void DrawLine(Pen, Point, Point);
- public void DrawLine(Pen, PointF, PointF);
- public void DrawLine(Pen, int, int, int, int);
- public void DrawLine(Pen, float, float, float, float);

Let's consider an example of drawing different geometric shapes. In Visual Studio, create a new **Windows Forms** project,

name it **DrawMetods**. Generate a method of the **Paint** event handler for the form, fill it with the below program logic.

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;

    //apply smoothing
    g.SmoothingMode = System.Drawing.Drawing2D.
            SmoothingMode.HighQuality;
    g.DrawLine(new Pen(Color.Red, 2), 0, 0, 100, 100);

    //draw a rectangle
    g.DrawRectangle(new Pen(Color.Green, 2),
                    new Rectangle(100, 100, 60, 60));

    //draw a pie
    g.DrawPie(new Pen(Color.Indigo, 3), 150, 10, 150, 150,
                    90, 180);

    //draw a text
    g.DrawString("Hello GDI+!", new Font("Verdana",
                12,FontStyle.Bold), Brushes.Black, 0, 240);

    //draw a polygon
    PointF[] pArray = {new PointF(10.0F, 50.0F),
            new PointF(200.0F, 200.0F),
            new PointF(90.0F, 20.0F),
            new PointF(140.0F, 50.0F),
            new PointF(40.0F, 150.0F)};
    g.DrawPolygon(new Pen(Color.GreenYellow, 2), pArray);

    //draw an ellipse
    g.DrawEllipse(new Pen(Color.Green, 4), 100, 230, 30, 30);
    g.Dispose();
}
```

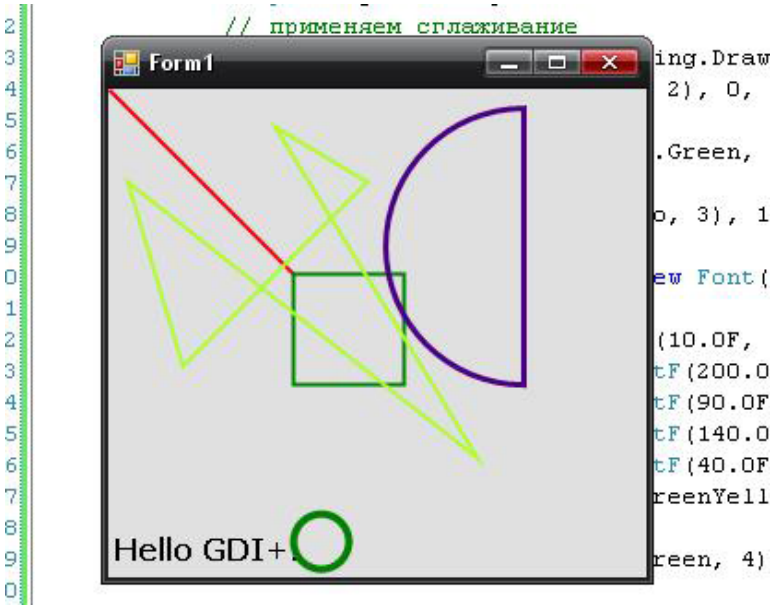

Fig. 6.2. Working with Graphics methods

7. Paint Event

Let's consider the following example in order to understand the work of the Paint event.

Launch Visual Studio, choose **File->New->Project**, specify Windows Forms Application as a type of the project, name it **PaintExample**, click ok.

In the **InitializeComponent()** method of the **Fosm1.Desiner.cs** file, which can be found in the Solution Explorer window, change the background color and window size.

```
private void InitializeComponent()
{
    this.SuspendLayout();
    //
    //Form1
    ...
    //, change background color and window size
    this.BackColor = System.Drawing.SystemColors.
        AppWorkspace;
    this.ClientSize = new System.Drawing.Size(400, 400);
}
```

Then add the following logic to form constructor of the Form1.cs file.

```
public Form1()
{
    InitializeComponent();
    this.Show();
    Graphics g = this.CreateGraphics();
    SolidBrush redBrush = new SolidBrush(Color.Red);
```

```
    Rectangle rect = new Rectangle(0, 0, 250, 140);
    g.FillRectangle(redBrush, rect);
}
```

Now run the application, the result will appear as shown in Figure 7.1.

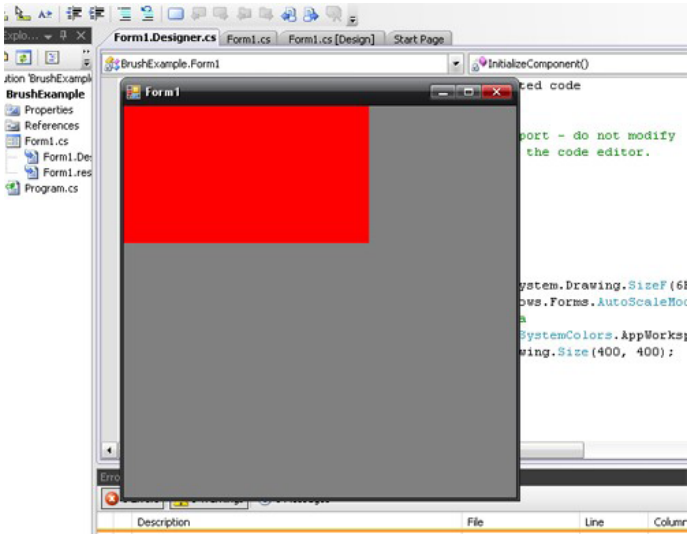

Fig. 7.1.

Now, try to minimize the window by pressing the button at the top left and restore it; we see that our red rectangle disappears and is no longer displayed. The problem also arises if we try to place another window under ours, our square is partially erased. Figure 7.2.

What happens? The fact is that Windows hides the invisible part of the window and thus frees up memory. On the desktop, we can open several dozen of windows. If Windows stored all the visual information, the video card memory would be heavily loaded.

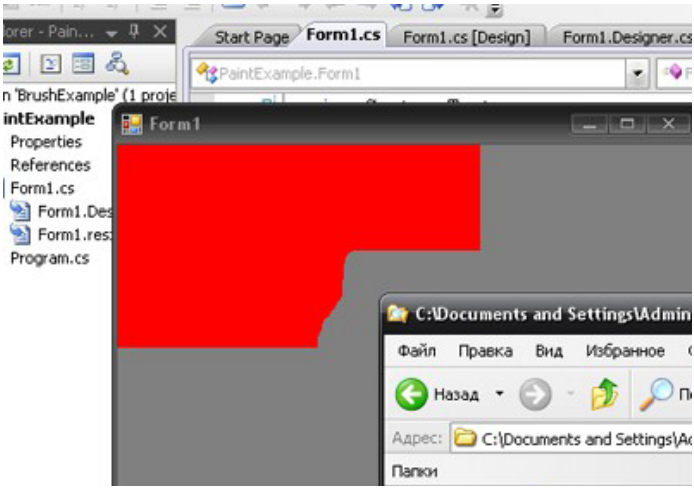

Fig. 7.2.

The problem is that we have placed the code that draws a rectangle in the constructor body and it actuates only once when the application is invoked. In order to solve this problem, our form needs the event that draws the form when it is needed.

To do this, Windows Forms has a **Paint** event; Windows invokes the event, the form produces drawing.

The above code is not correct and is presented only to demonstrate how the form drawing occurs.

Now, let's fix our code. Delete the entry, which was written in the constructor. Override the **OnPaint**() virtual method of the **Control** class.

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

```
    protected override void OnPaint(PaintEventArgs e)
    {
        base.OnPaint(e);
        Graphics g =e.Graphics;
        SolidBrush redBrush = new SolidBrush(Color.Red);
        Rectangle rect = new Rectangle(0, 0, 250, 140);
        g.FillRectangle(redBrush, rect);
    }
}
```

Run the application. As you can see, now, everything works fine and there are no problems that previously existed.

An alternative way to obtain the event method is as follows: in Visual Studio, go to the forms designer Form1.cs[Design]*, open the context menu using the right mouse button and select Properties, find the Paint event in this window and double click on it with the left mouse button. Figure 7.3.

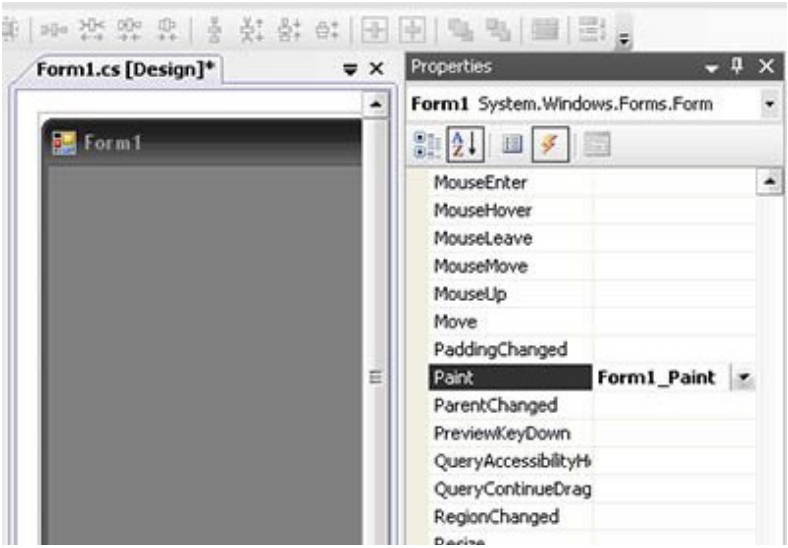

Fig. 7.3.

Note that the event handler method has been automatically generated.

```
public partial class Form1 : Form
{

    public Form1()
    {
        InitializeComponent();
    }

        private void Form1_Paint(object sender,
            PaintEventArgs e)
    {

    }
}
```

Now, fill it with the below program logic.

```
private void Form1_Paint(object sender, PaintEventArgs e)
{

    Graphics g = e.Graphics;
    SolidBrush redBrush = new SolidBrush(Color.Red);
    Rectangle rect = new Rectangle(0, 0, 250, 140);
    g.FillRectangle(redBrush, rect);

}
```

You need to remember that the **Paint** event is generated whenever form drawing is necessary.

Now, let's examine how and how many times a **Paint** event occurs, it fairly simple to do so. Add a counter, which will count the **Paint** events, to our code, and we can also observe when it occurs.

```
public partial class Form1 : Form
{
    private int paintCount;

    public Form1()
    {
        InitializeComponent();
        paintCount = 0;
    }
    ...
```

Initialize it in the constructor with a zero value; calculation is performed in the **Form1_Paint** method and the value is output on the form using the **Label** element.

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    ...
    this.label1.Text = String.Format("paintCount: {0}",
                        paintCount++);
}
```

Run the application. We see that at start-up the counter is equal to 1. In the Start menu, select Explorer and point to our form, note that the counter increases only when we move the Explorer down. Figure 7.4.

It is obvious that the **Paint** event actuates, when the Explorer overlaps any area of the form. The question is why do we need to redraw the entire form, if we need to redraw the red rectangle only?

GDI+ uses the terminology such as **Clipping Region**. It defines the place on the form where the redrawing will occur.

Using it you can notify the DC device context, what area needs to be redrawn.

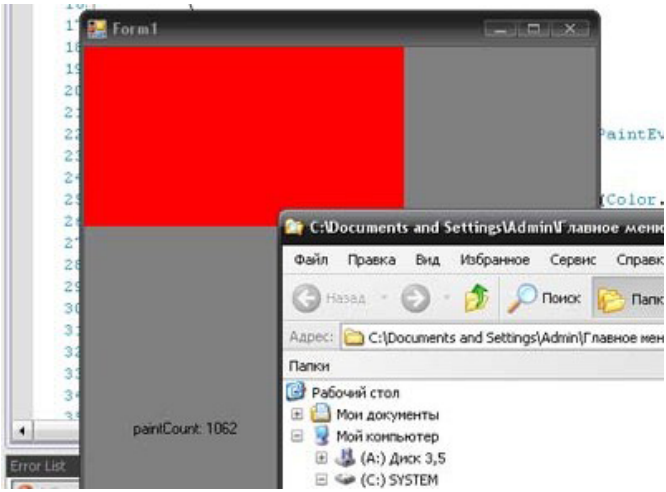

Fig. 7.4

PaintEventArgs argument of the Paint event can gain access to the clipping region via the **ClipRectangle** property. **ClipRectangle** returns a **System.Drawing.Rectangle** structure instance. The structure has four properties: **Top**, **Bottom**, **Left**, **Right**, describing vertical and horizontal coordinates.

Now let's change our code.

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    //using this condition, define a clipping region
    //140 and 250 are dimensions of our rectangle
    if (e.ClipRectangle.Top < 140
                        && e.ClipRectangle.Left < 250)
    {
        Graphics g = e.Graphics;
        SolidBrush redBrush = new SolidBrush(Color.Red);
```

```
            Rectangle rect = new Rectangle(0, 0, 250, 140);
            g.FillRectangle(redBrush, rect);

            this.label1.Text =
                String.Format("paintCount: {0}", paintCount++);
        }
}
```

Run the program. Note that the counter will increase only when the Explorer will be over our rectangle. We can say that we've just increased the performance of our application.

This project is called **PaintExample**.

8. Methods for displaying the simplest graphics primitives

8.1. Displaying a point

Let's consider an example of point construction on the client area. In the **Graphics** class there is no method for drawing a point. To draw a point, you can fill the inner area of geometric shapes, such as rectangle or ellipse and use methods such as **FillRectangle()** or **FillEllipse()** of the **Graphics** object.

Create a new Windows Forms project, name it **DrawPoints**. Generate a **Paint** event handler method for the form, fill it with the following program logic.

```
public partial class Form1 : Form
{
    List<Point> points = new List<Point>();
    public Form1()
    {
        InitializeComponent();
    }

    private void Form1_Paint(object sender,
            PaintEventArgs e)
    {
        Graphics g = e.Graphics;
        //draw all the points in the collection
        foreach (Point p in points) g.FillEllipse(Brushes.
            Black, p.X, p.Y, 10F, 10F);
```

```
    }
        private void Form1_MouseClick(object sender,
            MouseEventArgs e)
    {
        //add a new point to the collection
        points.Add(new Point(e.X, e.Y));
        //perform the redrawing of the client area
        Invalidate();
    }
}
```

In order a set of points is displayed on the client area, we create a collection, fill it by pressing the left mouse button. With the help of the **MouseEventArgs** argument of the **MouseClick** event handler method, pass the coordinates to the **Point** class constructor, a instance of which is passed to the **Add()** method, which fills the **points** collection. Redrawing of all the points is performed in the **Paint** event handler method.

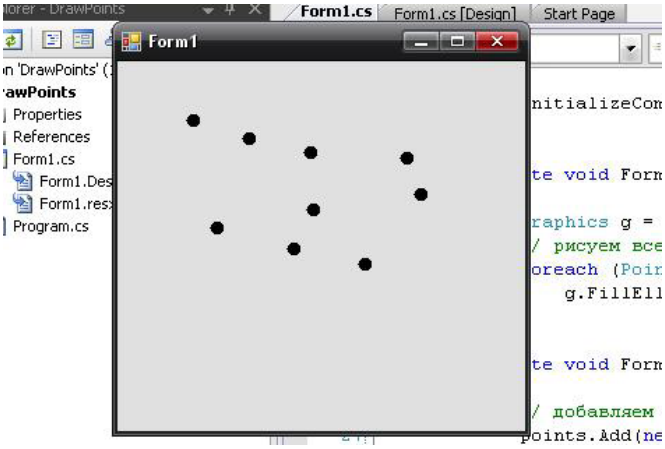

Fig. 8.1. Displaying a point.

The project is called **DrawPoints**.

8.2. Displaying a line

In previous chapters, you have already seen how to construct a line, we used the **DrawLine()** method of the **Graphics** object. Lines may have different styles. For example, we can draw a dot-dash line with an arrow at the end (**Cap**).

The line consists of three parts: a line body, a start and end **Cap**. A part, which connects the ends of the line, is called **Body**.

Fig. 8.2.

The line style is defined by the **Pen** class, through its properties and methods we can determine how the ends and body of the line will look like. For example, the **StartCap** property sets a style for the start point of the line, using the **DashStyle** one can specify a dotted pattern of the line body, **DashCap** allows you to define how the ends of dotted lines will be presented, for example: **Flat**, **Round** or **Triangle**.

Let's consider an example of constructing a stylized line. Create a new Windows Forms project, call it **DrawLine**. Add the **System.Drawing.Drawing2D** namespace. Generate a **Paint** event handler method for the form, fill it with the following program logic.

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.SmoothingMode = SmoothingMode.HighQuality;
    Pen bluePen = new Pen(Color.Blue, 6);
```

```
        //Set a style for ends and body of the line
        bluePen.StartCap = LineCap.SquareAnchor;
        bluePen.EndCap = LineCap.ArrowAnchor;
        bluePen.DashStyle = DashStyle.Dash;
        bluePen.DashCap = DashCap.Round;
        g.DrawLine(bluePen, 20, 100, 270, 100);
        bluePen.Dispose();
        g.Dispose();
}
```

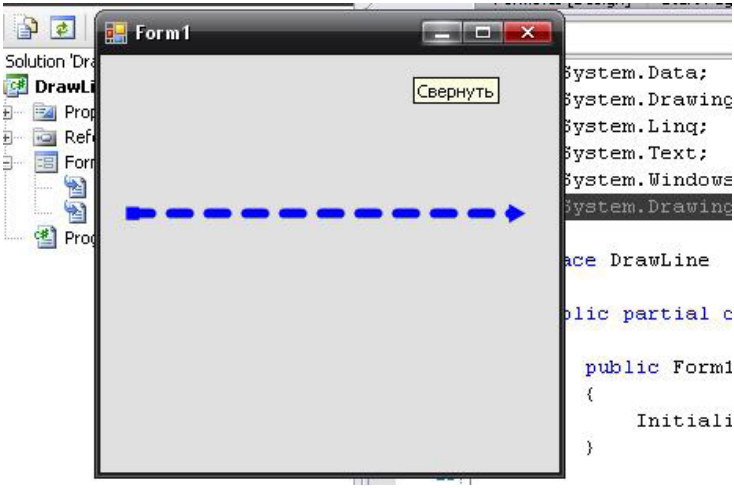

Fig. 8.3. Displaying a line.

The project is called **DrawLine**.

8.3. Displaying a rectangle

There are several ways to create a **Rectangle**. We can create a rectangle by passing four values to the **Rectangle** structure constructor as parameters, which are the starting point and size, or by passing such structures as **Point** and **Size**.

Rectangle can also be created by the **RectangleF** structure, which is a mirror reflection of the **Rectangle** structure, includes the same properties and methods, except that **RectangleF** receives floating-point values.

To display a rectangle in the client area, an instance of **Rectangle** must be passed as a parameter to the **DrawRectangle()** or **FillRectangle()** method of the **Graphics** object. The first parameter of the methods retrieves a **Pen** or **Brush** object, the second retrieves the **Rectangle** structure that represents a position and dimensions of the rectangle.

Let's consider an example of drawing a rectangle. Create a new Windows Forms project, name it **DrawRectangle**. Generate a **Paint** event handler method for the form, fill it with the following program logic

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
    int x = 20;
    int y = 30;
    int height = 60;
    int width = 60;
    //create a starting point
    Point pt = new Point(10, 10);
    //Create size
    Size sz = new Size(160, 140);
    //create two rectangles
    Rectangle rect1 = new Rectangle(pt, sz);
    Rectangle rect2 = new Rectangle(x, y, width, height);
    //drawing the rectangles
    g.FillRectangle(Brushes.Black, rect1);
    g.DrawRectangle(new Pen(Brushes.Red, 2), rect2);
}
```

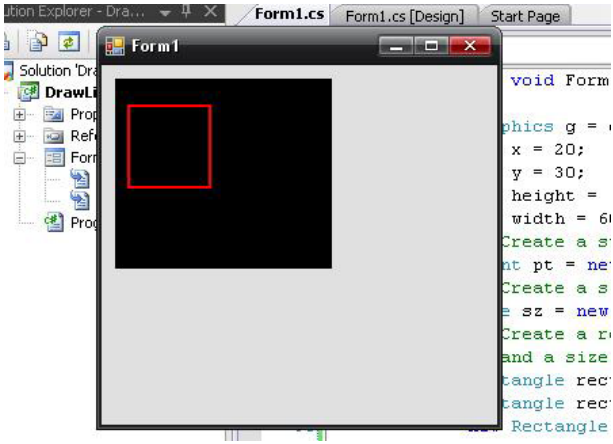

Fig. 8.4. Displaying a rectangle.

The project is called **DrawRectangle**.

8.4. Displaying an ellipse

As a rectangle, an ellipse can be displayed in several ways. Drawing is made by **DrawEllipse()** or **FillEllipse()** methods of the **Graphics** object. The first parameter of the methods retrieves a **Pen** or **Brush** object, the second retrieves the **Rectangle** structure that represents a position and dimensions of the ellipse.

Let's consider an example of drawing an ellipse. Create a new Windows Forms project, name it **DrawEllipse**. Generate a **Paint** event handler method for the form, fill it with the following program logic

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
    int x = 23;
    int y = 33;
```

```
        int height = 60;
        int width = 60;
        Pen pn = new Pen(Brushes.Red, 4);
        //create a starting point
        Point pt = new Point(10, 10);
        //create size
        Size sz = new Size(160, 160);
        //create two rectangles
        Rectangle rect1 = new Rectangle(pt, sz);
        Rectangle rect2 = new Rectangle(x, y, width, height);
        g.FillEllipse(Brushes.Black, rect1);
        g.DrawEllipse(pn,rect2);
    }
```

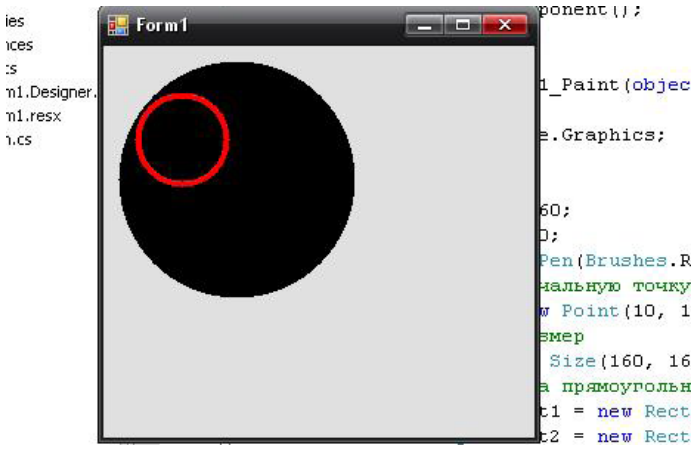

Fig. 8.5 Displaying an ellipse.

The project is called **DrawEllipse**.

9. Color, Size, Rectangle, Point Structures

9.1. Color Structure

In GDI+, colors are represented by instances of **System.Drawing.Color** structure.

The first method to create a **Color** structure is to specify the values for red, green and blue colors, causing a **Color.FromArgb()** aggregate function.

```
Color pink = Color.FromArgb(241, 105, 190);
```

Three parameters are respectively the amounts of red, blue and green. There are several other overloaded methods for this function; some of them also allow you to define the so-called alpha-blend. Alpha blending allows you to draw translucent tones, combining with a color that already exists on the screen that can create beautiful effects and is often used in games.

Creating a structure using **Color.FromArgb()** is the most flexible technique, since it in fact means that any color, which can be distinguished by the human eye, can be determined. However, if you want a simple, standard, well-known color such as red or blue, it is much easier just to name a desired color. In this regard, Microsoft provides a large number of statistical properties for **Color**, each one returning a named color. There are several hundreds of such colors. Full list is filed in the MSDN documentation. It includes all the simple colors: **Red**, **Green**, **Black**, etc., as well as such as **DarkOrchid**, **LightCoral** and so on.

```
Color navy = Color.Navy;
Color silver = Color.Silver;
Color sringGreen = Color.SpringGreen;
```

The Color structure has the following methods:

Name	Description
`public static Color FromArgb(int argb)`	Creates a Color structure from a 32-bit ARGB value.
`public static Color FromArgb(int alpha, Color baseColor)`	Creates a Color structure from the specified Color structure, but with the new specified alpha value. Although this method allows a 32-bit value to be passed for the alpha value, the value is limited to 8 bits. Valid alpha values for a new Color are 0 through 255.
`public static Color FromArgb (int red, int green, int blue)`	Creates a Color structure from the specified 8-bit color values (red, green, and blue). The alpha value is implicitly 255 (fully opaque). Although this method allows a 32-bit value to be passed for each color component, the value of each component is limited to 8 bits.
`public static Color FromArgb(int alpha,int red,int green,int blue)`	Creates a Color structure from the four ARGB component (alpha, red, green, and blue) values. Although this method allows a 32-bit value to be passed for each component, the value of each component is limited to 8 bits.
`public static Color FromKnownColor (KnownColor color)`	Creates a Color structure from the specified predefined color.

Name	Description
`public static Color FromName (string name)`	Creates a Color structure from the specified name of a predefined color.
`public static Color FromSysIcv (int icv)`	Gets a system-defined color. An integer used to retrieve the system color.
`public float GetBrightness()`	Gets the hue-saturation-brightness (HSB) brightness value for this Color structure.
`public float GetHue()`	Gets the hue-saturation-brightness (HSB) hue value, in degrees, for this Color structure.
`public float GetSaturation()`	Gets the hue-saturation-brightness (HSB) saturation value for this Color structure.
`public int ToArgb()`	Gets the 32-bit ARGB value of this Color structure.
`public KnownColor ToKnownColor()`	Gets the KnownColor value of this Color structure.
`public override string ToString()`	Converts this Color structure to a human-readable string

Operators

Name	Description
`public static bool operator == (Color left, Color right)`	Tests whether two specified Color structures are equivalent.
`public static bool operator != (Color left, Color right)`	Tests whether two specified Color structures are different.

Fields

Name	Description
`public static readonly Color Empty`	Represents a color that is a null reference.

Properties

Name	Description
`public byte A { get; }`	Gets the alpha component value of this Color structure.
`public byte B { get; }`	Gets the blue component value of this Color structure.
`public byte G { get; }`	Gets the green component value of this Color structure.
`public bool IsEmpty { get; }`	Specifies whether this Color structure is uninitialized.
`public bool IsKnownColor { get; }`	Gets a value indicating whether this Color structure is a predefined color. Predefined colors are represented by the elements of the KnownColor enumeration.
`public bool IsNamedColor { get; }`	Gets a value indicating whether this Color structure is a named color or a member of the KnownColor enumeration.
`public bool IsSystemColor { get; }`	Gets a value indicating whether this Color structure is a system color. A system color is a color that is used in a Windows display element. System colors are represented by elements of the KnownColor enumeration.
`public string Name { get; }`	Gets the name of this Color.
`public byte R { get; }`	Gets the red component value of this Color structure.

9.2. Size Structure

In GDI+, the Size structure is used to represent dimensions in pixels. It specifies both a height and width.

Constructor of this structure is overloaded:

Name	Description
`public Size(Point pt)`	Initializes a new instance of the **Size** class from the specified **Point** object.
`public Size(int width, int height)`	Initializes a new instance of the **Size** class from the specified dimensions.

Methods

Name	Description
`public static Size Add(Size sz1, Size sz2)`	Adds the width and height of one Size structure to the width and height of another **Size** structure.
`public static Size Ceiling(SizeF value)`	Converts the specified **SizeF** structure to a **Size** structure by rounding the values of the **Size** structure to the next higher integer values.
`public override bool Equals(Object obj)`	Tests whether the specified object is a **Size** structure with the same dimensions as this **Size** structure.
`protected virtual void Finalize()`	Allows an **object** to try to free resources and perform other cleanup operations before it is reclaimed by garbage collection.
`public static Size Round(SizeF value)`	Converts the specified **SizeF** structure to a **Size** structure by rounding the values of the **SizeF** structure to the nearest integer values.

Name	Description
`public static Size Subtract(Size sz1, Size sz2)`	Subtracts the width and height of one Size structure from the width and height of another Size structure.
`public override string ToString()`	Creates a human-readable string that represents this Size structure.
`public static Size Truncate(SizeF value)`	Converts the specified SizeF structure to a Size structure by truncating the values of the SizeF structure to the next lower integer values.

Operators

Name	Description
`public static Size operator +(Size sz1, Size sz2)`	Adds the width and height of one Size structure to the width and height of another Size structure.
`public static bool operator == (Size sz1, Size sz2)`	Tests whether two Size structures are equal.
`public static explicit operator Point (Size size)`	Converts the specified Size to a Point.
`public static implicit operator SizeF (Size p)`	Converts the specified Size to a SizeF.
`public static bool operator != (Size sz1, Size sz2)`	Tests whether two Size structures are different.
`public static Size operator – (Size sz1, Size sz2)`	Subtracts the width and height of one Size structure from the width and height of another Size structure.

Fields

Name	Description
`public static readonly Size Empty`	Gets a new instance of the Size class.

Properties

Name	Description
`public int Height { get; set; }`	Gets or sets the vertical component of this Size structure.
`public bool IsEmpty { get; }`	Tests whether this Size structure has width and height of 0.
`public int Width { get; set; }`	Gets or sets the horizontal component of this Size structure

9.3. Point Structure

In GDI+, the **Point** structure is used to describe a separate point, located in a two-dimensional plane. It is passed as an argument to many functions used in GDI+, such as **DrawLine()**. Constructors are defined for the structure:

```
int x = 15, y = 20;
Point p0 = new Point();
Point p1 = new Point(x);
Point p2 = new Point(x, y);
Point p3 = new Point(new Size(x, y));
```

The following methods are defined in the **Point** structure:

Name	Description
`public static Point Add(Point pt, Size sz)`	Adds the specified Size to the specified Point.
`public static Point Ceiling(PointF value)`	Converts the specified PointF to a Point by rounding the values of the PointF to the next higher integer values.
`public override bool Equals(Object obj)`	Specifies whether this Point contains the same coordinates as the specified Object.

Name	Description
`protected virtual void Finalize()`	Allows an object to try to free resources and perform other cleanup operations before it is reclaimed by garbage collection.
`public void Offset(Point p)`	Translates this **Point** by the specified **Point**.
`public void Offset(int dx, int dy)`	Translates this **Point** by the specified amount(dx , dy)
`public static Point Round(PointF value)`	Converts the specified **PointF** to a **Point** object by rounding the **Point** values to the nearest integer.
`public static Point Subtract(Point pt, Size sz)`	Returns the result of subtracting specified **Size** from the specified **Point**.
`public override string ToString()`	Converts this **Point** to a human-readable string.
`public static Point Truncate(PointF value)`	Converts the specified **PointF** to a **Point** by truncating the values of the **Point**.

Operators

Name	Description
`public static Point operator + (Point pt, Size sz)`	Translates a **Point** by a given **Size**.
`public static bool operator == (Point left, Point right)`	Compares two **Point** objects. The result specifies whether the values of the X and Y properties of the two **Point** objects are equal.
`public static explicit operator Size (Point p)`	Converts the specified **Point** structure to a **Size** structure.
`public static implicit operator PointF (Point p)`	Converts the specified **Point** structure to a **PointF** structure.

Name	Description
`public static bool operator != (Point left, Point right)`	Compares two **Point** objects. The result specifies whether the values of the X or Y properties of the two **Point** objects are unequal.
`public static Point operator (Point pt, Size sz)`	Translates a **Point** by the negative of a given **Size**.

Fields

Name	Description
`public static readonly Point Empty`	Represents a **Point** that has X and Y values set to zero.

Properties

Name	Description
`public bool IsEmpty { get; }`	Gets a value indicating whether this **Point** is empty.
`public int X { get; set; }`	Gets or sets the x-coordinate of this **Point**.
`public int Y { get; set; }`	Gets or sets the y-coordinate of this **Point**.

9.4. Структура Rectangle

In GDI+, this structure is used to set coordinates of a rectangle. The **Point** structure describes the top-left corner of the rectangle, and the **Size** structure — its dimensions. **Rectangle** has two constructors. X, Y coordinates, width and height passed as arguments to first one. The second constructor takes the **Point** and **Size** structures.

```
int x = 15, y = 20, h = 70, w = 200;
Rectangle rect0 = new Rectangle(x, y, w, h);
//or
Rectangle rect1 = new Rectangle(new Point(x, y),
                  new Size(w, h));
```

Methods

Name	Description
`public static Rectangle Ceiling(RectangleF value)`	Converts the specified **RectangleF** structure to a **Rectangle** structure by rounding the **RectangleF** values to the next higher integer values.
`public bool Contains(Point pt)`	Determines if the specified point is contained within this **Rectangle** structure.
`public bool Contains(Rectangle rect)`	This method returns **true** if the rectangular region represented by rect is entirely contained within this **Rectangle** structure; otherwise **false**.
`public bool Contains(int x, int y)`	Determines if the specified point is contained within this **Rectangle** structure.
`public override bool Equals(Object obj)`	Tests whether obj is a **Rectangle** structure with the same location and size of this **Rectangle** structure.
`protected virtual void Finalize()`	Allows an object to try to free resources and perform other cleanup operations before it is reclaimed by garbage collection.
`public static Rectangle FromLTRB (int left, int top, int right, int bottom)`	Creates a **Rectangle** structure with the specified edge locations.

Name	Description
`public void Inflate (Size size)`	Enlarges this **Rectangle** by the specified size value.
`public void Inflate (int width, int height)`	Enlarges this **Rectangle** by the specified width and height values.
`public static Rectangle Inflate (Rectangle rect, int x, int y)`	Creates and returns an enlarged copy of the specified **Rectangle** structure. The copy is enlarged by the specified amount. The original **Rectangle** structure remains unmodified.
`public void Intersect(Rectangle rect)`	Replaces this **Rectangle** with the intersection of itself and the specified **Rectangle**.
`public static Rectangle Intersect(Rectangle a, Rectangle b)`	Returns a third **Rectangle** structure that represents the intersection of two other **Rectangle** structures. If there is no intersection, an empty **Rectangle** is returned.
`public bool IntersectsWith (Rectangle rect)`	Determines if this rectangle intersects with rect.
`public void Offset(Point pos)` `public void Offset(int x, int y)`	This method adjusts the location of the upper-left corner horizontally by the x-coordinate of the specified point, and vertically by the y-coordinate of the specified point.
`public static Rectangle Round (RectangleF value)`	Converts the specified **RectangleF** to a **Rectangle** by rounding the **RectangleF** values to the nearest integer values.
`public override string ToString()`	Converts the attributes of this **Rectangle** to a human-readable string.

Name	Description
`public static Rectangle Truncate (RectangleF value)`	Converts the specified **RectangleF** to a **Rectangle** by truncating the **RectangleF** values.
`public static Rectangle Union (Rectangle a, Rectangle b)`	Gets a **Rectangle** structure that contains the union of two **Rectangle** structures.

Operators

Name	Description
`public static bool operator == (Rectangle left, Rectangle right)`	This operator returns **true** if the two **Rectangle** structures have equal X, Y, Width, and Height properties.
`public static bool operator != (Rectangle left, Rectangle right)`	This operator returns **true** if any of the X, Y, Width or Height properties of the two **Rectangle** structures are unequal; otherwise **false**.

Fields

Name	Description
`public static readonly Rectangle Empty`	Represents a **Rectangle** structure with its properties left uninitialized.

Properties

Name	Description
`public int Bottom { get; }`	Gets the y-coordinate that is the sum of the Y and Height property values of this **Rectangle** structure.
`public int Height { get; set; }`	Gets or sets the height of this **Rectangle** structure.
`public bool IsEmpty { get; }`	Tests whether all numeric properties of this **Rectangle** have values of zero.

Name	Description
`public int Left { get; }`	Gets the x-coordinate of the left edge of this **Rectangle** structure.
`public Point Location { get; set; }`	Gets or sets the coordinates of the upper-left corner of this **Rectangle** structure.
`public int Right { get; }`	Gets the x-coordinate that is the sum of X and Width property values of this **Rectangle** structure.
`public Size Size { get; set;}`	Gets or sets the size of this **Rectangle**.
`public int Top { get; }`	Gets the y-coordinate of the top edge of this **Rectangle** structure.
`public int Width { get; set; }`	Gets or sets the width of this **Rectangle** structure.
`public int X { get; set; }`	Gets or sets the x-coordinate of the upper-left corner of this **Rectangle** structure.
`public int Y { get; set; }`	Gets or sets the y-coordinate of the upper-left corner of this **Rectangle** structure.

10. Brushes

10.1. Drawing2D space

The **System.Drawing.Drawing2D** namespace provides advanced two-dimensional and vector graphics functionality. It provides the possibility to install special "caps" for pens, to create brushes that create not a continuous line, but textures, to produce a variety of vector manipulations with graphic objects.

Class category	Details
Графика и графические контуры	The GraphicsState and GraphicsContainer classes report information about the current Graphics object. GraphicsPath classes represent a series of lines and curves. The GraphicsPathIterator and PathData classes provide detailed information about the contents of a GraphicsPath object.
Типы, относящиеся к матрице и преобразованию	The Matrix class represents a matrix for geometric transforms. The MatrixOrder enumeration specifies the order for matrix transformations.
Классы Brush	The PathGradientBrush and HatchBrush classes enable you to fill shapes with either a gradient, or hatch pattern, respectively.
Перечисление, связанное с линиями	The LineCap and CustomLineCap enumerations enable you to specify cap styles for a line. The, LineJoin enumeration enables you to specify how two lines are joined in a path.

Class category	Details
	The PenAlignment enumeration enables you specify the alignment of the drawing tip, when you draw a line. The PenType enumeration specifies the pattern a line should be filled with.
Перечисления, связанные с заполнением фигур и контуров	The HatchStyle enumeration specifies fill styles for a HatchBrush. The Blend class specifies a blend pattern for a LinearGradientBrush. The FillMode enumeration specifies the fill style for a GraphicsPath.

10.2. Brush Class

Brushes are designed for "painting" the space between the lines. You can define a color, texture or image for the brush. The Brush itself is an abstract class, and you cannot create objects of this class. The **Brush** class is in the **System.Drawing** namespace. **TextureBrush**, **HatchBrush** and **LinearGradientBrush** derived classes are in **System.Drawing.Drawing2D** namespace. Let's consider the **Brush** class descendants in more detail.

10.3. SolidBrush Class

SolidBrush fills a shape with a solid color. The **Color** property allows getting or setting the color of a **SolidBrush** object.

10.4. TextureBrush Class TextureBrush

TextureBrush allows filling a shape with a picture stored in the binary representation. This class cannot be inherited. When creating such a brush it is also required to set a framing

rectangle and a frame mode. The framing rectangle determines what portion of the picture we have to use when drawing, it is not needed to use the entire picture. Here are some members of this class:

Name	Description
`public Image Image { get; }`	Gets the Image object associated with this TextureBrush object.
`public Matrix Transform {get; set;}`	Gets or sets a copy of the Matrix object that defines a local geometric transformation for the image associated with this TextureBrush object.
`public WrapMode WrapMode {get; set;}`	Gets or sets a WrapMode enumeration that indicates the wrap mode for this TextureBrush object.

Methods

Name	Description
`public void MultiplyTransform (Matrix matrix)` `public void MultiplyTransform (Matrix matrix, MatrixOrder order)`	Multiplies the Matrix object that represents the local geometric transformation of this TextureBrush object by the specified Matrix object by prepending the specified Matrix object.
`public void ResetTransform()`	Resets the Transform property of this TextureBrush object to identity.
`public void RotateTransform (float angle)`	Rotates the local geometric transformation of this TextureBrush object by the specified amount. This method prepends the rotation to the transformation.

Name	Description
`public void RotateTransform (float angle, MatrixOrder order)`	Rotates the local geometric transformation of this TextureBrush object by the specified amount in the specified order.
`public void ScaleTransform (float sx, float sy)` `public void ScaleTransform (float sx, float sy, MatrixOrder order)`	Scales the local geometric transformation of this TextureBrush object by the specified amounts. This method prepends the scaling matrix to the transformation. **sx** — The amount by which to scale the transformation in the x direction. **sy** — The amount by which to scale the transformation in the y direction. **order** — A MatrixOrder enumeration that specifies whether to append or prepend the scaling matrix.
`public void TranslateTransform (float dx, float dy)` `public void TranslateTransform (float dx, float dy, MatrixOrder order)`	Translates the local geometric transformation of this TextureBrush object by the specified dimensions in the specified order. **dx** — The dimension by which to translate the transformation in the x direction. **dy** — The dimension by which to translate the transformation in the y direction. **order** — The order (prepend or append) in which to apply the translation.

10.5. HatchBrush Class

More complex "painting" can be made by using a derivative **HatchBrush** of the **Brush** class. This type allows you to paint the interior region of the object by means of a large number of hatching specified in the **HatchStyle** enumeration (there

are 54 unique hatch styles). The main color sets the color of the lines; background color defines the color of intervals between lines. This class cannot be inherited.

Name	Description
`public Color BackgroundColor { get; }`	Gets the color of spaces between the hatch lines drawn by this HatchBrush object.
`public Color ForegroundColor { get; }`	Gets the color of hatch lines drawn by this HatchBrush object.
`public HatchStyle HatchStyle { get; }`	Gets the hatch style of this HatchBrush object.

10.6. LinearGradientBrush Class

LinearGradientBrush contains a brush that allows you to draw a smooth blend from one color to another, herewith the first color blends to the second one at a certain angle. Wherein angles are specified in degrees. The angle of 0˚ means that the blend from one color to another is from left to right. The angle of 90 ˚ means that the blend from one color to another is carried out from top to bottom. This class cannot be inherited. Here are some properties of **LinearGradientBrush**:

Properties

Name	Description
`public Blend Blend { get; set; }`	Gets or sets a Blend that specifies positions and factors that define a custom falloff for the gradient.
`public bool GammaCorrection { get; set; }`	Gets or sets a value indicating whether gamma correction is enabled for this LinearGradientBrush.

Name	Description
`public ColorBlend InterpolationColors { get; set; }`	Gets or sets a ColorBlend that defines a multicolor linear gradient.
`public Color[] LinearColors { get; set; }`	Gets or sets the starting and ending colors of the gradient.
`public RectangleF Rectangle { get; }`	Gets a rectangular region that defines the starting and ending points of the gradient.

10.7. PathGradientBrush Class

PathGradientBrush allows creating a complex shading effect, which uses a color change from the middle of the path being drawn to its edges.

Name	Description
`public Color CenterColor { get; set; }`	Gets or sets the color at the center of the path gradient.
`public PointF CenterPoint { get; set; }`	Gets or sets the center point of the path gradient.
`public PointF FocusScales { get; set; }`	Gets or sets the focus point for the gradient falloff.
`public ColorBlend InterpolationColors { get; set; }`	Gets or sets a ColorBlend that defines a multicolor linear gradient.
`public Color [] SurroundColors { get; set; }`	Gets or sets an array of colors that correspond to the points in the path this PathGradientBrush fills.

11. Pen class

Typical application of **Pen** objects is to draw lines. Usually, a **Pen** object is not used by itself: it is passed as a parameter to multiple output methods defined in the **Graphics** class.

This class provides several overloaded constructors, with which you can specify the source color and the thickness of the pen. Most of the **Pen** features are determined by the properties of this class.

Properties (selectively)

Name	Description
`public PenAlignment Alignment {get; set;}`	This property determines how the **Pen** draws closed curves and polygons. The **PenAlignment** enumeration specifies five values; however, only two values — **Center** and **Inset** — will change the appearance of a drawn line. **Center** is the default value for this property and specifies that the width of the pen is centered on the outline of the curve or polygon. A value of Inset for this property specifies that the width of the pen is inside the outline of the curve or polygon. The other three values, **Right**, **Left**, and **Outset**, will result in a pen that is centered. A **Pen** that has its alignment set to Inset will yield unreliable results, sometimes drawing in the inset position and sometimes in the centered position. Also, an inset pen cannot be used to draw compound lines and cannot draw dashed lines with **Triangle** dash caps.

Name	Description
`public Brush Brush { get; set; }`	Gets or sets the **Brush** that determines attributes of this **Pen**.
`public float[] CompoundArray { get; set; }`	Gets or sets an array of values that specifies a compound pen. A compound pen draws a compound line made up of parallel lines and spaces.
`public CustomLineCap CustomStartCap { get; set; }`	Gets or sets a custom cap to use at the beginning of lines drawn with this **Pen**.
`public DashStyle DashStyle { get; set; }`	Gets or sets the style used for dashed lines drawn with this **Pen**.
`public PenType PenType { get; }`	Gets or sets the style of lines drawn with this **Pen**.

DashStyle Enumeration

Member name	Description
`Solid`	Specifies a solid line.
`Dash`	Specifies a line consisting of dashes.
`Dot`	Specifies a line consisting of dots.
`DashDot`	Specifies a line consisting of a repeating pattern of dash-dot.
`DashDotDot`	Specifies a line consisting of a repeating pattern of dash-dot-dot.
`Custom`	Specifies a user-defined custom dash style.

In GDI+, in addition to the **Pen** class you can also use a collection of pre-defined pens (**Pens** collection). Using the static properties of the **Pens** collection you can instantly get

a ready pen, without having to create it manually. However, all types of **Pen**, which are created using the **Pens** collection, have the same width equal to 1. It cannot be changed.

LineCap enumeration is used to work with the "caps" of pens.

Values of the **LineCap** enumeration

Value	Description
`ArrowAnchor`	Lines end with arrowheads.
`DiamondAnchor`	Lines end with diamonds
`Flat`	Standard rectangular line caps
`Round`	Line caps are rounded
`RoundAnchor`	Lines with a round anchor cap.
`Square`	Lines end with squares of line thickness
`SquareAnchor`	Lines end with squares of higher thickness than that of lines.
`Triangle`	Triangular line cap.

It is advisable to call the **Dispose()** method for the **Pen** and **Brush** objects being created, or you should use the **using** construction to work with them, otherwise the application can run out of system resources.

12. Application examples of brushes and pens

Let's consider the example of a program using brushes and pens of various kinds. Create a static Cub class, implementing the creation of a graphical cube and different methods of filling the faces. The BrushesExampleMethod(Graphics g) shows the work with the brushes. Let's create a Color structure using the Color.FromArgb() method:

```
Color pink = Color.FromArgb(241, 105, 190);

//create a solid brush of pink color
SolidBrush sldBrush = new SolidBrush(pink);

//using a sldBrush fill a rectangle, the upper left and
//corner of which has coordinates of (300;150),
//height and width of 70
g.FillRectangle(sldBrush, 300, 150, 70, 70);
```

Let's consider the use of hBrush hatching. As a constructor parameters let's pass a hatching style, line color and color of the gaps between the lines:

```
HatchBrush hBrush = new HatchBrush(HatchStyle.
    NarrowVertical, Color.Pink, Color.Blue);
g.FillRectangle(hBrush, 370, 150, 70, 70);
```

Create an lgBrush gradient brush. The Rectangle structure represents boundaries of a linear gradient:

```
LinearGradientBrush lgBrush = new LinearGradientBrush(new
    Rectangle(0, 0, 20, 20), Color.Violet,
    Color.LightSteelBlue, LinearGradientMode.Vertical);
g.FillRectangle(lgBrush, 300, 220, 70, 70);
```

To fill the next rectangle we will use a built-in brush from the **Brushes** collection:

```
g.FillRectangle(Brushes.Indigo, 370, 220, 70, 70);
```

Let's use the texture brush. Load a pattern for the brush from the file:

```
TextureBrush tBrush = new TextureBrush(Image.
        FromFile(@"Images\charp.jpg"));
g.FillRectangle(tBrush, 370, 290, 70, 70);
```

In the **DrawLeftAndUpperBound** method of the **Cub** class, draw the left and top faces of the cube:

```
public static void DrawLeftAndUpperBound(Graphics g)
{
Point[] p = { new Point(240, 110), new Point(440, 110),
            new Point(510, 150), new Point(300, 150) };
TextureBrush tBrush = new TextureBrush(Image.
            FromFile(@"Images\0073.jpg"));
g.FillPolygon(tBrush, p);

Point[] p1 = { new Point(240, 110), new Point(300, 150),
            new Point(300, 360), new Point(240, 310) };
HatchBrush hBrush = new HatchBrush(HatchStyle.
            DashedDownwardDiagonal,
            Color.Violet, Color.White);
g.FillPolygon(hBrush, p1);
```

Work with pens is shown in the **PensExampleMethod** method of the Cub class :

```
public static void PensExampleMethod(Graphics g)
{
//draw vertical lines with the solid pen
    Pen p = new Pen(Color.White, 1);
    for (int i = 0; i < 4; i++)
    {
        g.DrawLine(p,300+70*i,150,300+70*i,360);
    }
//draw horizontal lines with the dashed pen
    p = new Pen(Color.White, 1);
    p.DashStyle = DashStyle.DashDot;
    for (int i = 0; i < 4; i++)
    {
        g.DrawLine(p, 300, 150 + 70 * i, 510, 150 + 70 * i);
    }
//draw oblique lines using the pen with diamond cap
    p = new Pen(Color.White, 1);
    p.StartCap = LineCap.DiamondAnchor;
    p.EndCap = LineCap.DiamondAnchor;
    for (int i = 0; i < 3; i++)
    {
        g.DrawLine(p, 240+70*i, 110 , 300+70*i, 150);
    }
}
```

Home task

1. Create an application that will draw a chessboard and chess pieces on the client area of the form. A context menu should be displayed for each figure.
2. Create an application that allows drawing different geometric shapes on the client area. In the top, the application should have a toolbar with buttons that allow you to enter the settings of the geometric primitive and draw them. Drawn primitives are to be displayed on the client area of the form. Also, add the ability to save songs on the hard drive in any format.