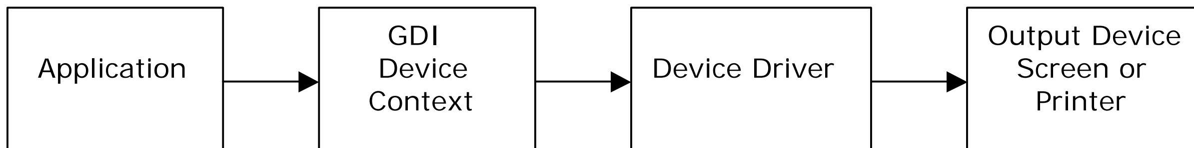
GDI+ A Higher Level API

The **Graphics device interface** (GDI) provides windows applications with a device independence means that the program is able to work using different screens, keyboards and printers without any modification to the program. Example, If an application prints for an Epson dot matrix printer then it also printes for a Laser printer without changing the application.

The GDI is a layer between the application and the different types of hardware. This architecture frees the programmer from having to deal with each type of device directly by letting the GDI resolve differences in the hardware instead. The Graphics device interface is a part of Windows that converts the windows graphics to the actual commands sent to the hardware. The GDI is a program file stored on your computer. The windows environment loads GDI into memory when it is required for graphical output. Windows also loads a device driver program if the hardware conversions are not a part of GDI. A device driver is a program that assists the GDI in converting windows graphics commands to hardware commands. Refer figure below for GDI scenario.



Device Context : The basic tool that windows uses to provide device independence for an application is called a device context or DC. The DC is an internal structure that windows uses to maintain information about an output device. A device context is a link between a windows application, a device driver and an output device such as printer or plotter.

GDI is a set of C++ classes, which provides functionality to render data to a program to hardware devices with the help of device drivers. GDI sits between the program and the hardware and transfer data from one to other. In Visual Studio .NET, Microsoft has taken care of most of the GDI problems and have made it easy to use. **The GDI version of .NET is called GDI+.**

GDI+ is next evolution of GDI. It's much better improved and easy to use version of GDI. The best thing about GDI is you don't need to know any details of drivers to render data on printers and monitors. GDI+ takes care of it for you. In other words, GDI was a low-middle level of programming API, where you need to know about devices too, while GDI+ is a higher level of programming model, which provides functions to do work for you. For example, if you want to set background or foreground color of a control, just set BackColor or ForeColor property of the control.

Changes in programming model (GDI vs GDI+)

GDI uses the concept of Device Context (DC). Every device context is associated with a window. To draw on a window, one must first obtain a device context of that window. To change any attribute, say, pen colour, it first has to be selected in the device context by calling the SelectObject( ) method. Once selected, all the drawing is done using this pen, until such time as another pen is selected in device context.

GDI+ works with **graphics context** that plays a similar role as device context. The graphics context is also associated with a particular window and contains information

specifying how a drawing would be displayed. However, unlike device context, it does not contain information about pen, brush, font, etc. In order to draw with a new pen we simply have to pass an object of Pen class to the DrawLine( ) method (this method draws a line on window). We can pass different Pen objects in each call to DrawLine( ) method to draw the lines in different colours. Thus **GDI uses a stateful model, whereas GDI+ uses a stateless model**. The Graphics class encapsulates the graphics context. Most of the drawing is done by calling methods of the Graphics class

Beside the fact that GDI+ API is easier and flexible than GDI, there are many more new features added to the API.

Some of the new features GDI+ offers are –

?? Improved Colors. Now GDI+ comes with more colors and these are compatible with other colors such as Windows etc.
?? Antialiasing support
?? Gradient brushes
?? Splines
?? Transformation and Matrices
?? Scalable reasons
?? Alpha Blending

GDI+ Class and Interfaces in .NET

In Microsoft .NET library, all classes (types) are grouped in namespaces. A **namespace** is nothing but a category of similar kind of classes. For example, Forms related classes are stored in Windows.Forms namespace. Similarly, GDI+ classes are grouped under six namespaces, which reside in System.Drawing.dll assembly.

GDI+ Namespaces :

GDI+ is defined in the Drawing namespace and its five sub namespaces.

All drawing code resides in **System.Drawing.DLL** assembly. These namespaces are System.Drawing,

System.Drawing.Design,
System.Drawing.Printing,
System.Drawing.Imaging,
System.Drawing.Drawing2D and
System.Drawing.Text.

System.Drawing Namespace :

The **System.Drawing** namespace provides basic GDI+ functionality. If contains the definition of basic classes such as Brush, Pen, Graphics, Bitmap, Font etc. The Graphics class plays a major role in GDI+ and contains methods for drawing to the display device. The following table contains some of the System.Drawing namespace classes, structures and their definition.

Classes

Classes	Description
Bitmap, Image	Bitmap and image classes.
Brush, Brushes	Brush classes used define objects to fill GDI objects such as rectangles, ellipses, pies, polygons, and paths.

Font, FontFamily	Defines a particular format for text, including font face, size, and style attributes. Not inheritable.
Graphics	Encapsulates a GDI+ drawing surface. Not inheritable.
Pen	Defines an object used to draw lines and curves. Not inheritable.
SolidBrush	Defines a brush of a single color. Brushes are used to fill graphics shapes, such as rectangles, ellipses, pies, polygons, and paths. Not inheritable.

Structures

Structure	Description
Color	Represents an ARGB color.
Point, PointF	Represents a 2D x- and y-coordinates. Point takes x, y values as a number. You can use PointF if you want to use floating number values.
Rectangle, RectangleF	Represents a rectangle with integer values. A rectangle represents two point pair – top, left and bottom, right. You can use floating values in RectangleF.
Size	Size of a rectangular region with an ordered pair of width and height. Size takes an integer as width and height while SizeF takes floating numbers to represent width and height.

System.Drawing.Design Namespace :

The **System.Drawing.Design** namespace is somewhat smaller in compare to the System.Drawing. It extends design-time user interface (UI) logic and drawing functionality and provides classes for customizing toolbox and editor classes. For beginners there is nothing in this namespace. It has two types of classes –

Editor Classes

BitmapEditor, FontEditor, and ImageEditor are the editor classes. You can use these classes to extend the functionality and provide an option in properties window to edit images and fonts.

ToolBox Classes

ToolBoxItem, ToolBoxItemCollection are two major toolbox classes. By using these classes you can extend the functionality of toolbox and provide the implementation of toolbox items.

System.Drawing.Drawing2D Namespace :

This namespace consists classes and enumerations for advanced 2-dimmensional and vector graphics functionality. It contains classes for gradient brushes, matrix and transformation and graphics path. Some of the common classes and enumerations are defined in the following tables -

Classes

Class	Description
Blend and ColorBlend	These classes define the blend for gradient brushes. The ColorBlend defines array of colors and position for multi-color gradient.

GraphicsPath	This class represents a set of connected lines and curves.
HatchBrush	A brush with hatch style, a foreground color, and a background color.
LinearGradientBrush	Provides a brush functionality with linear gradient.
Matrix	3x3 matrix represents geometric transformation.

Enumerations

Enumeration	Description
CombineMode	Different clipping types
CompositingQuality	The quality of compositing
DashStyle	The style of dashed lines drawn with a Pen.
HatchStyle	Represents different patterns available for HatchBrush
QualityMode	Specifies the quality of GDI+ objects.
SmoothingMode	Specifies the smoothing quality of GDI+ objects.

System.Drawing.Imaging Namespace

This namespace provides advanced GDI+ imaging functionality. It defines classes for metafile images. Other classes are encoder and decoder, which let you use any image format. It also defines a class PropertyItem, which let you store and retrieve information about the image files.

System.Drawing.Printing Namespace

The **System.Drawing.Printing** namespace defines classes for printing functionality in your applications. Some of its major classes are defines in the following table -

Classes

Class	Description
PageSettings	Page settings
PaperSize	Size of a paper.
PreviewPageInfo	Print preview information for a single page.
PrintController	Controls document printing
PrintDocument	Sends output to a printer.
PrinterResolution	Sets resolution of a printer.
PrinterSettings	Printer settings

System.Drawing.Text Namespace

Even though most of the font's functionality is defined in System.Drawing namespace, this provides advanced typography functionality such as creating collection of fonts. Right now,

this class has only three classes – FontCollection, InstalledFontCollection, and PrivateFontCollection. All of these classes are self-explanatory

The Graphics Class

The **Graphics** class is center of all GDI+ classes. The Graphics class plays a vital role in GDI+. No matter where you go, you go through this class. The Graphics class encapsulates GDI+ drawing surfaces. Before drawing any object (for example circle, or rectangle) we have to create a surface using Graphics class.

There're different ways to get a graphics object in your application. You can either get a graphics object on your form's paint event or by overriding OnPaint() method of a form. These both have one argument of type **System.Windows.Forms.PaintEventArgs.** You call its Graphics member to get the graphics object in your application. For example

```
protected override void OnPaint(object obj, System.Windows.Forms.PaintEventArgs pea)
{
  Graphics g= pea.Graphics
}
```

Now you've the Graphics object. Now you can do any thing you want. The Graphics class has many methods for drawing graphics objects such as fonts, pens, lines, path and polygons, images and ellipse and so on. Some of the graphics class members are described in the following table below–

DrawArc	This method draws an arc.
DrawBezier, DrawBeziers, DrawCurve	These methods draw a simple and bazier curves. These curvers can be closed, cubic and so on.
DrawEllipse	Draws an ellipse or circle.
DrawImage	Draws an image.
DrawLine	Draws a line.
DrawPath	Draws the path (lines with GraphicsPath)
DrawPie	Draws the outline of a pie section.
DrawPolygon	Draws the outline of a polygon.
DrawRectangle	Draws the outline of a rectangle.
DrawString	Draws a string.
FillEllipse	Fills the interior of an ellipse defined by a bounding rectangle.
FillPath	Fills the interior of a path.
FillPie	Fills the interior of a pie section.
FillPolygon	Fills the interior of a polygon defined by an array of points.
FillRectangle	Fills the interior of a rectangle with a Brush
FillRectangles	Fills the interiors of a series of rectangles with a Brush
FillRegion	Fills the interior of a Region.

Now, if you want to draw an ellipse using the same method we've described above, you override OnPaint method and write the following code –

```
protected override void OnPaint( object obj, PaintEventArgs pea )
{
        Graphics g = pea.Graphics;
        Pen MyPen = new Pen(Color.Green, 10);
        g.DrawLine( MyPen, 10, 10, 200, 100);
        g.DrawEllipse(new Pen(Color.Red, 20), 10, 20, 100, 200);
}
```

Common Graphics Objects

The graphics objects are the objects, which you use to draw your GDI+ items such as images, lines, rectangles, and path. For example, to fill a rectangle with a color, you need a color object and type of style you want to fill such as solid, texture and so on.

There are four common GDI+ objects, which you'll be using throughout your GDI+ life to fill GDI+ items. The major objects are:

Brush : Used to fill enclosed surfaces with patterns, colors or bitmaps.

Pen : Used to draw lines and polygons including rectangles, arcs and pies.

Font : Used to describe the font to be used to render text.

Color : Used to describe the color used to render a particular object. In GDI+ color can be alpha blended.

The Pen Class

A pen draws a line of specified width and style. You always use Pen constructor to create a pen. The constructor initializes a new instance of the **Pen** class. You can initialize it with a color or brush.

Initializes a new instance of the Pen class with the specified color.

Pen (Color clr)

Initializes a new instance of the Pen class with the specified Brush.

Pen (Brush br)

Initializes a new instance of the **Pen** class with the specified Brush and width.

Pen (Brush br, float fWidth)

Initializes a new instance of the **Pen** class with the specified Color and Width.

Pen(Color clr , float fWidth)

Here is one example:

Pen MyPen = new Pen(Color.Blue);

or
Pen MyPen = new Pen(Color.Blue, 100);

Some of its most commonly used properties are:

Alignment : Gets or sets the alignment for objects drawn with this Pen.

Brush : Gets or sets the Brush that determines attributes of this **Pen**.

Color : Gets or sets the color of this **Pen**.

Width : Gets or sets the width of this **Pen**.

The Color Structure

A Color structure represents an ARGB color. Here are ARGB properties of it:

Pen MyPen= new Pen (Color.Blue);

A Gets the alpha component value for this Color.

B Gets the blue component value for this Color.

G Gets the green component value for this Color.

R Gets the red component value for this Color.

You can call the Color members. Each color name (say Blue) is a member of the Color structure. Here is how to use a Color structure:

Pen MyPen = new Pen(Color.Blue);

The Font Class

The **Font** class defines a particular format for text such as font type, size, and style attributes. You use font constructor to create a font.

Initializes a new instance of the **Font** class with the specified attributes.

Font(string strFamily, float fSizeinPoints)

Initializes a new instance of the **Font** class from the specified existing **Font** and FontStyle.

Font(Font f, FontStyle s);

Where FontStyle is an enumeration and here are its members:

Member Name	Description
Bold	Bold Text
Italic	Italic Text
Regular	Normal Text
StrikeOut	Text with a line through the middle
Underline	Underlined Text.

There are many overloaded constructors. Here is one example:
Font fnt = new Font(“Times New Roman”, 10);

Some of its most commonly used properties are:

Bold	Gets a value indication whether this Font is bold.
FontFamily	Gets the FontFamily of this Font.
Height	Gets the height of this Font.
Italic	Gets a value indication whether this Font is Italic.
Name	Gets the face name of this Font.
Size	Gets the size of this Font.
SizeInPoints	Gets the Size in points of this Font
StrikeOut	Gets a value indicating whether this Font is strikeout.
Style	Gets style information for this Font.
Underline	Gets a value indicating whether this Font is underlined.
Unit	Gets the unit of measure for this Font.

The Brush Class

The **Brush** class is an abstract base class and cannot be instantiated. We always use its derived classes to instantiate a brush object, such as SolidBrush, TextureBrush and LinearGradientBrush.

Here is one example:

```
LinearGradientBrush lbrush = new LinearGradientBrush(rect, Color.Red, Color.Yellow,
LinerGradientMode.BackwardDialognal);
```

Or

```
SolidBrush brh = new SolidBrush(Color.Red);
```

The HatchBrush class defines a brush that fills an area with small repeating pattern, most commonly consisting of horizontal, vertical or diagonal lines.

The SolidBrush class defines a brush made up of a single color. Brushes are used to fill graphics shapes such as rectangles, ellipses, pies, polygons, and paths.

The TextureBrush encapsulates a Brush that uses and fills the interior of a shape with an image that repeats horizontally and vertically.

The LinearGradiantBrush encapsulates both two-color gradients and custom multi-color gradients.

The Rectangle Structure

```
Rectangle( Point pt, Size sz);
Or
Rectangle( x1 : integer, y2 : integer, x3 : integer, x4 : integer);
```

The **Rectangle** structure is used to draw a rectangle. Besides its constructor, the Rectangle structure has following members:

Bottom	Gets the y-coordinate of the lower-right corner of the rectangular region defined by this Rectangle.
Height	Gets or sets the width of the rectangular region defined by this Rectangle.
IsEmpty	Tests whether this Rectangle has a Width or a Height of 0
Left	Gets the x-coordinate of the upper-left corner of the rectangular region defined by this rectangle
Location	Gets or sets the coordinates of the upper-left corner of the rectangular region represented by this rectangle

Right	Gets the x-coordinate of the lower-right corner of the rectangular region defined by this rectangle.
Size	Gets or sets the size of this Rectangle
Top	Gets the y-coordinate of the upper-left corner of the rectangular region defined by this Rectangle
Width	Gets or sets the width of the rectangular region defined by this rectangle
X	Gets or sets the x-coordinate of the upper left corner of the rectangular region defined by this rectangle
Y	Gets or sets the y-coordinate of the upper left corner of the rectangular region defined by this rectangle.

The Point Structure

This structure is similar to the POINT structure in C++. It represents an ordered pair of x and y coordinates that define a point in a two-dimensional plane. The member x represents the x coordinates and y represents the y coordinates of the plane.

Here is how to instantiate a point structure:

```
Point pt1 = new Point(30, 30);
```

Some sample Examples :

Adding reference to the namespaces

Before using GDI+ classes in your application, you need to add the corresponding namespace reference to the application. In my application, I use the following namespace to the application.

```
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
```

Invalidate Method

The Invalidate() method plays a major role in the case when you need to call OnPaint method programmatically. In our application, we call Invalidate in many cases.

Draw a rectangle

You can override OnPaint event of your form to draw an rectangle. The LinearGradientBrush encapsulates a brush and linear gradient.

```
Rectangle rect = new Rectangle(50, 30, 100, 100);

LinearGradientBrush lBrush = new LinerGradientBrush(rect, Color.Red, Color.Yellow,
LinerGradientMode.BackwardDiagonal);

g. FillRectangle(lBrush, rect);
```

Drawing an Arc

DrawArc function draws an arc. This function takes four arguments.

First is the Pen. You create a pen by using the **Pen** class. The Pen constructor takes at least one argument, the color or the brush of the pen. Second argument width of the pen or brush is optional.

```
Pen pn = new Pen(Color.Blue);
```

Or

```
Pen pn = new Pen(Color.Blue, 100);
```

The second argument is a rectangle. You can create a rectangle by using Rectangle structure. The Rectangle constructor takes four int type arguments and they are left and right corners of the rectangle.

```
Rectangle rect = new Rectangle( 50, 50, 200, 100 );
```

Now,

```
Pen MyPen = new Pen( Color.Blue );
Rectangle rect = new Rectangle( 50, 50, 200, 100);
g.DrawArc( MyPen, rect, 12, 84 );
```

Drawing a Line

DrawLine function of the Graphics class draws a line. It takes three parameters, a pen, and two Point class parameters, starting and ending points. Point class constructor takes x, y arguments.

```
protected override void OnPaint( PaintEventArgs pea )
{
	Graphics g = pea.Graphics;
	Pen Mypen = new Pen( Color.Blue );
	Point pt1 = new Point( 30, 30 );
	Point pt2 = new Point( 110, 100 );
	g.DrawLine( Mypen, pt1, pt2 );
}
```

Draw an Ellipse

An ellipse(or a circle) can be drawn by using DrawEllipse method. This method takes only two parameters, Pen and rectangle.

```
protected override void OnPaint( PaintEventArgs pea )
{
	Graphics g = pea.Graphics;
	Pen MyPen = new Pen( Color.Blue, 100);
	Rectangle rect = new Rectangle( 50, 50, 200, 100 );
	g.DrawEllipse( MyPen, rect );
}
```

Gradient

```
protected override void OnPaint( PaintEventArgs pea )
{
	Graphics g = pea.Graphics;
	Rectangle rect = new Rectangle( 50, 30, 200, 200 );
	LinearGradientBrush brush= new LinearGradientBrush( rect, Color.Blue, Color.Green,
	LinearGradientMode.Vertical );
	g.FillRectangle( brush, rect );
}
```

Drawing a Text

You can override OnPaint event of your form to draw a text on the form.

```
protected  override void OnPaint( PaintEventArgs pea)
{
```

```
        Graphics g = pea.Graphics;
        g.DrawString( "Hello Universe", this.Font, new SolidBrush(Color.Red), 20, 20);
}
```

Paths Region and Clipping

Path provides a way to connect straight lines and curves.

Clipping is the restriction of graphics output to a particular area of screen.

Region describes an area of the output device in device coordinates.

A path is a collection of device-independent cordinate points that describe straight lines and curves. These lines and curves might or might not be connected to each other. Any set of connected lines and curves within a path is known as a figure or a subpath. Thus a path is composed of zero or more subpaths. Each subpath is a collected of connected lines and curves. A subpath can be either open or closed.

Normally you can create a new path using the default constructor

```
GraphicsPath path = new GraphicsPath();
```

Now you can call methods of GraphicsPath class that add straight lines and curves to the path.

Here are the GraphicsPath methods :

```
void AddLine(... )
void AddLines(... )
void AddArc(... )
void AddBezier(... )
void AddBeziers(... )
void AddCurve(... )
```

if path is an object of type GraphicsPath the following three call add three connected lines to the path

```
path.AddLine(0, 0, 0, 100);
path.AddLine(100, 100, 100, 0);
```

because the first line ends at (0, 100) and second line begins at (100, 100) the path adds a line between those two points.

Start and Close methods include :

```
void StartFigure()
void CloseFigure()
void CloseAllFigures()
```

All three of these calls end the current subpath and begin a new subpath and CloseFigure closes the current subpath.

Other GraphicsPath Methods include

```
void AddRectangle(... )
void AddRectangles(... )
void AddPolygon(... )
```

void AddEllipse(...)
void AddPie(...)
void AddClosedCurve(...)
void AddPath(GraphicsPath path, bool bconnect)

for Rendering the path Graphics Methods are

void DrawPath(Pen pen, GraphicsPath path)
void FillPath(Brush brush, GraphicsPath path)

The DrawPath method draws the lines and curves that comprise the path using the specified pen. FillPath fills the interior of all closed subpaths using the specified brush.

Besides drawing and filling paths, we can also use paths to set a **clipping** region for the Graphics object.

void SetClip(GraphicsPath path)
void SetClip(GraphicsPath path, CombineMode cm)

void SetClip(Rectangle rect)
void SetClip(Rectangle rect, CombineMode cm)
void SetClip(RectangleF rectf)
void SetClip(RectangleF rectf, CombineMode cm)

void ResetClip()

The graphics class also includes methods named IntersectClip and ExcludeClip to modify the existing clipping region. To return the clipping region to normal call ResetClip() method.

Region describes an area of the output device. When you define a path for clipping, the path is converted into a region. Only one of the constructors of Region class creates a region directly from a path.

Region(GraphicsPath path)

The region encompasses the interiors of all the subpaths in the path. If the subpaths have overlapping areas, the filling mode of the path determines which interior areas become part of the region and which ones do not. Only one method uses a region for drawing.

void FillResion(Brush brush, Region rgn)

and Only one version of the SetClip method uses a region directly

void SetClip(Region rgn, CombineMode cm)

GDI+ Example :

Here is a small program that draws graphics on the form using GDI+.

Create a new windows forms application.

A window receives WM_PAINT message when it is to be painted. We need to handle this message if we want to do any painting in the window. In .NET we can do this either by overriding the virtual method OnPaint( ) of the Form class or by writing a handler for the Paint event.

Add the Paint handler to the form. The MyPaintEventHandler( ) handler would look like this.

```
private void MyPaintEventHandler (object sender, PaintEventArgs e)
{
}
```

The first parameter passed to the MyPaintEventHandler handler contains the reference to the object of a control that sends the event. The second parameter contains more information about the Paint event.
Now lets display a string on the form. To display the string we use DrawString() method of the Graphics Class.

```
private void MyPaintEventHandler(object sender, PaintEventArgs pea)
{
        Graphics g = pea.Graphics;
        Font myfont = new Font ("Arial", 23);
        StringFormat fmt = new StringFormat();
        fmt.Alignment = StringAlignment.Center;
        fmt.LineAlignment = StringAlignment.Center;
        g.DrawString ("Hello Universe!", myfont, Brushes.Blue, ClientRectangle, fmt);
}
```

Here PaintEventArgs is a class and contains reference to the Graphics object. We can use this reference for drawing.

In a handler other than Paint event handler we can obtain the Graphics reference using the CreateGraphics( ) method of the Form class.

The DrawString() method has several overloaded versions. We have used one that allows us to display centrally aligned text in desired font and colour.

The first parameter passed to the DrawString( ) method is the string to display.
The second parameter is the font in which text would get displayed. We have created a font by passing the font name and font size to the constructor of the Font class.
The text gets filled with the brush colour specified as the third parameter.
The fourth parameter specifies the surrounding rectangle. We have passed ClientRectangle property of Form class that contains a rectangle representing the client area of the form.
To centrally align the text we have used the StringFormat class. The Alignment and LineAlignment properties of this class contain horizontal and vertical alignment of text respectively.

The Graphics class contains various methods to draw different shapes. This includes drawing rectangle, line, arc, bezier, curve, pie, etc.

The following code draws a rectangle using green coloured pen having line thickness of 3.

```
Pen p=new Pen(Color.Green, 3);
g.DrawRectangle(p, 20, 20, 150, 100);
```

The Pen class encapsulates various styles of pens like solid, dash, dash-dot, etc. We can change the style of pen using the DashStyle property of the Pen class. This is shown in the following statement.

```
p.DashStyle = DashStyle.Dash;
```

If we want, we can specify custom pen style by using the DashPattern property. There are several other properties of the Pen class that allow us to specify the pen type (hatch fill, gradient fill, solid color, etc), cap style, join style, etc.

Unlike GDI, GDI+ provides separate methods for rectangle and filled rectangle. To fill the rectangle we need to pass a Brush object. This is shown below.

```
HatchBrush hb=new HatchBrush(HatchStyle.BackwardDiagonal,
Color.Red, Color.Black);
g.FillRectangle (hb, 0, 0, 250, 200);
```

We have used hatch brush to fill the rectangle. The hatch brush is created using the HatchBrush class. hatch style as BackwardDiagonal. The rectangle will get filled with the hatch brush in a red and black colour combination.

Like the HatchBrush class there are several other classes used to fill the shapes with, namely, SolidBrush, TextureBrush, and LinearGradientBrush. Gradient brush is something that was not available in GDI.

```
LinearGradientBrush gb=new LinearGradientBrush(ClientRectangle,
Color.Blue, Color.Red, 90);
g.FillRectangle(gb,ClientRectangle);
```

Here, we have created an object of the LinearGradientBrush class and passed to its constructor the rectangle to be filled, and two colours that form the gradient pattern. The last parameter specifies the angle from which we wish to draw. Specifying 90 would fill the window vertically.

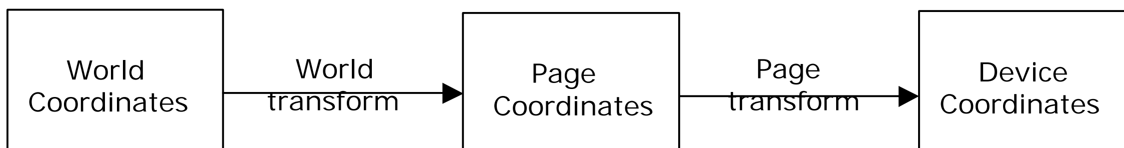
Coordinates and Transformations

A transformation is an algorithm that alters ("transforms") the size, orientation, and shape of objects. Page-Space to Device-Space Transformations. A mapping mode is a scaling transformation that specifies the size of the units used for drawing operations. The mapping mode may also perform translation.

Properties of Graphics

DpiY vertical resolution in dots per inch
DpiX horizontal resolution in dots per inch

In the Graphics methods we specify coordinates in a two-dimensional coordinate system. The system has the origin at the top-left corner and x and y axes point to the right and down respectively. All the methods take coordinates in pixels. The coordinates passed to Graphics methods are world coordinates. When we pass world coordinates to a method, they first gets translated into page coordinates (logical coordinates) and then into device coordinates (physical coordinates).



Ultimately, the shape gets drawn in device coordinates. In both the page and device coordinate system the measure of unit is the same—pixels. The coordinate system can be customised by shifting the origin to some other place in the client area and by setting a different measure of unit. Let see how to use it. We would first draw a horizontal line having 1 inch of width.

```
private void MyPaintEventHandler(object sender, PaintEventArgs pea)
{
        Graphics g=pea.Graphics;
        g.PageUnit=GraphicsUnit.Inch;
        Pen p=new Pen (Color.Green, 1/g.DpiX);
        g.DrawLine (p, 0, 0, 1, 0);
}
```

Here, firstly we have set the PageUnit property to GraphicsUnit.Inch specifying that the unit of measure is an inch. We have created a Pen object and set its width to 1 / g.Dpix. The Dpix property of the Graphics class indicates a value, in dots per inch, for the horizontal resolution supported by this Graphics object. Note that this is necessary because now Pen object also assumes 1 unit = 1 inch. So, if we don't set the pen width like this, a line with 1 inch pen width would get drawn. Next we draw a line having one unit measure, which happens to be an inch.

Let us now shift the origin to the centre of the client area and draw the line again.

```
private void MyPaintEventHandler (object sender, PaintEventArgs pea)
{
        Graphics pea = e.Graphics;
        g.PageUnit = GraphicsUnit.Inch;
        g.TranslateTransform ( ( ClientRectangle.Width/g.DpiX )/ 2,
        (ClientRectangle.Height/g.DpiY)/2);
        Pen p = new Pen ( Color.Green, 1/g.DpiX);
        g.DrawLine (p, 0, 0, 1, 0);
}
```

Here, after setting the unit to an inch using the PageUnit property, we have called the TranslateTransform( ) method to shift the origin to the centre of the client area. This method maps the world coordinates to page coordinates and so the transformation is called world transformation. The x and y values we have passed to the TranslateTransform( ) method get added to every x and y values we pass to the Graphics methods. Finally, we created a pen having proper width and draw the line.

GDI+ also allows us to orient the x and y axes' direction to the specified angle. For this, it provides the RotateTransform( ) method. For example, if we call the RotateTransform( ) method before drawing the line as shown below,

```
g.RotateTransform(30);
```

then line would get displayed slanting downwards, 30 degrees below the base line. We can use this functionality of the RotateTransform( ) method to create an application like an analogue clock.

Disposing graphics objects

Whenever we open a file, we close it after we have finished working with the file. This is because a handle is associated with the file and it remains open if we don't close it explicitly. Similarly, GDI+ resources like pens, brushes and fonts need to be disposed of because they encapsulate GDI+ handles in them. To release the GDI+ resources, we can call the Dispose( )

method on every object that is to be released. For example, the following statement would release the pen object represented by pn using the Dispose( ) method.

```
pn.Dispose();
```

Release all the Graphics object obtained by calling the CreateGraphics() method.