Chapter 3
Forms, Text, and Graphics – A Closer Look

In this chapter we will explore forms, text, and graphics in greater depth. This is essential if you want to write more serious applications. I could try to cover every detail of these topics, but that would take many pages and you normally don’t need to know the advanced features except for a particular application. Instead we will cover the knowledge that applies to almost every program you write.

Form Properties

The Form class is central to Windows Forms programming and we need to take a closer look at this class since we inherit all its fields, properties, events, and methods in the form we design with the Visual Studio 2005 designer. Properties are extremely important since we modify many aspects of the form using its properties. When we change a property using the properties window we are actually either modifying or adding code to our program. As we discussed in the last chapter this code is normally hidden from your view in the partial class file managed by the designer. There is one important difference. Properties set with the designer or properties window are set when your application starts up. These are initial values. What if we want to change a property at some point during the programs execution, perhaps as a result of a button click. In other object oriented programming languages we use fields or data members to store what we would regard as a property of the class. Of course we use fields to hold other information associated with the class. You can think of a property as an aspect of the class that relates to things like the look and feel of a form. For example, the background color or title of the form. A better understanding of properties in the C# language is important at this time.

The C# Property

A C# property consists of methods to get and set the property. The actual property can be any legal type or types. Normally these are kept in fields with the private access attribute. That’s so we can’t directly manipulate the values, but must instead use the get and set special methods. Prior to C# programmers quite often wrote a pair of public methods to access the private data associated with the property. Here are the statements needed to implement a simple integer property:
The keyword `value` is used to access the value being assigned to the property. Now we can assign to, and assign from the `publicProperty` property we have defined.

```csharp
int i;
publicProperty = 1000;
i = publicProperty;
```

After these three statements execute the variable `i` will have the value 1000. You should notice that `publicProperty` is used just as if it was a variable. The fact that method calls are actually being made is hidden. That's a very powerful feature of the C# property.

You might be asking why all this is necessary since making the integer `hiddenProperty` public would server the same purpose. In this case it would. However, there are many situations where we need to execute some statements as well as setting or returning the hidden data. We might want to verify that a value is within bounds. We might want the internal representation of the property to be different than the public representation. A very important action that we might want for a form is to have it repaint if we change the property. Changing a simple field will not accomplish this. This will become a bit clearer when we take a look at some important properties of the `Form` class. Here is a simple example that changes the form's background color in response to a button click:

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

namespace ChangeBackgroundColor
{
    public partial class Form1 : Form
    {
```
I placed a button on the form and used the properties window to change the text to "Change Color" and to set up a click event handler. When the button is clicked not only does the background color change to black, but the text in the button changes to "Black." Figure 3-1 and Figure 3-2 show the before and after appearance of the form. Notice that the background color of the button does not change. Every control has its own properties and the background color of the button is distinct from the background color of the form.
Color is a structure and not a class. It contains a large number of static properties that evaluate to a Color object. We can also create a custom color if one of these colors is not exactly what you want, but I will defer discussion of this until further along.

When you invoked the set method of the BackColor property this caused the form to be repainted with the new background color. The button was also repainted on top of our new background color as well. Without the capabilities of the C# property we would have had to make a method call to Invalidate or use some other method to cause the Windows Paint event to be generated. The C# property is a language feature that you should not underestimate.

**Overriding the OnPaint Method**

Thus far we have been implementing the Paint event by providing our own handler. There is actually a preferred technique. The Form class has a protected OnPaint method. This is actually an override from the OnPaint method that is inherited from the Control class. We can override this method in our derived class. When we override a method in C# it means that our method is called instead of the method in the base class. The OnPaint method in the Form class is called in response to the Paint event by default. If we override this method in our derived class then our method will be called rather than the method in the base class. This is the format of the override:

```csharp
protected override void OnPaint(PaintEventArgs e)
{
}
```
As you can see we have a PaintEventArgs parameter just like with the Paint event handler. We do everything just like we would if we implemented our own event handler. It is strongly suggested that you call the base class method to make sure that other registered delegates receive the event. In other words, if we have other event handlers associated with the Paint event we want to make sure that they are called as well. Here is how we do it:

```csharp
protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e);
    ...
}
```

The base keyword is used to indicate that we want to call the method in the base class. If you forget this keyword you will make a recursive call to yourself and run out of stack. Your program will appear to lock up and eventually the error in Figure 3-3 will appear. You must run your program under the debugger to capture such errors. Use Debug|Start Debugging from the menu or just hit the F5 key to run your program in under the debugger. It is a good idea to always run your program under the debugger until you know it runs without errors. Visual Studio 2005 is amazing in that it even makes the suggestion that you may have an infinite recursion in your program.

I will use the override of OnPaint in future programs.

---

**StackOverflowException was unhandled**

An unhandled exception of type 'System.StackOverflowException' occurred in MouseClick.EXE

Troubleshooting tips:

- Make sure you do not have an infinite loop or infinite recursion.
- Get general help for this exception.

Search for more Help Online...

Actions:

- View Detail...
- Copy exception detail to the clipboard

![Figure 3-3](image)

**The Form's Title Property**
A form normally has a title bar at the top. This can be set to give a name for application or something else. Microsoft Word 2003 displays the name of the file for the currently viewed document or a generic name if it hasn't been saved to disk yet. You can set the title using the properties window or even programmatically during the programs execution. The Text property corresponds to the title for a Form. Changing the Text property for a button, as you recall, changed the label on the button. The following code sets the title in the form's constructor:

```csharp
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        this.Text = "Title Demo Project";
    }
}
```

Figure 3-4 shows the result when we execute the program. Functionally there is little difference in using the designer to set the title or the constructor in this case. However, let's say you wanted to display the current date in the title. Unless you want to rebuild your project every day you need set the title at execution time. Since we only need to do it once, the code can be placed in the constructor.

Figure 3-4

**The Form's Position and Size**
Three properties control the size and position of the form.

Form Properties
Type Property
Size Size
Point Location
FormStartPosition StartPosition

The *Size* type is very similar to the *Point* type except that it has an integer *Width* and *Height* rather than an *X* and *Y* coordinate. *Location* specifies the position of the upper left hand corner of the form relative to the upper left hand corner of the screen. *FormStartPosition* is an enumeration.

Form Start Position
Member Action
CenterParent Centers the form in the parent form (if exists).
CenterScreen Centers the form on the screen.
Manual Location and Size properties are used.
WindowsDefaultBounds Size and position determined by Windows
WindowsDefaultLocation Position only is determined by Windows

For complete control of the size and position of your form set the *StartPosition* to *Manual* in the properties window or using this statement:

```csharp
this.StartPosition = FormStartPosition.Manual;
```

One other property is related to the position of the form. The *WindowState* property controls whether the form is initially displayed maximized, minimized or using the default position. If you set this property to *Maximized*, for example, the form will fill the entire screen regardless of the *Size* and *Location* properties.

We will discuss some other properties when we discuss dialogs. A dialog is just a form with some properties changed. Among these are the *FormBorderStyle*, *MaximizeBox* and *MinimizeBox* properties.

**Scrolling a Form**

There are a number of ways to scroll a form's client area. There are some neat tricks that can be used to make it easier. To appreciate these tricks let's take a look at the basic issue of scrolling. We need to add scrolling to a form if the amount of information we want to display exceeds the available area available on the screen. This is obvious for applications like text editors where many lines of text need to be available to the user. Unfortunately the Windows operating system does not have the capability to automatically scroll the client area of a window. There is actually an operating system method available that
will scroll the contents of a window by moving all the pixels up or down. However, it has no way of painting the area that becomes exposed. Anyone who has programmed in the early days of Windows may remember that scrolling was complex. Things did not get much better with the introduction of Application Frameworks such as the Microsoft Foundation Class Library, MFC.

If all the content of our form consists of controls such as buttons we can add scroll bars to a form quite easily. Unfortunately this technique doesn't work if we paint the form using the Graphics class. To scroll a form with just controls all we need to do is set the form's Autoscrol property to true. Both horizontal and vertical scroll bars appear if the form is too small to display all the controls. In this example I have placed four buttons on the form, one in each corner. Figure 3-5 shows what the form looks like when all controls are within the size of the form. If the form is resized such that not all the controls are completely visible both horizontal and vertical scroll bars appear as shown in Figure 3-7 and Figure 3-7. We can use the scroll bars to view the hidden buttons.

![Form1](image-url)
So what do we do if we are painting directly to the form’s client area? The first step is to determine the size of the client area you would need to display all of your output. If you are positioning graphics by specific coordinates you should be able to do this by using the rightmost and bottommost graphic. If you are displaying text things get a little more complicated. I will discuss issues of determining the size of text when we take a closer look at the `DrawString` method. Instead let's use a simple example that essentially duplicates the button example. The following `OnPaint` method results in the output of Figure 3-8.

```csharp
protected override void OnPaint(PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.Gray, 10, 10, 75, 23);
    g.FillRectangle(Brushes.Gray, 200, 10, 75, 23);
```
The coordinate of the lower right hand corner rectangle's lower right hand corner is (275, 223) since each rectangle is 75 x 23 pixels (the dimensions of a default button) and this rectangle is painted at (200, 200). Let's add a little extra space to bring the width and height of the required client area to 300 x 250.

The next step is to set the `AutoScrollMinSize` property to this dimension. You can do this with the form's properties window. If the scrollable window's size will change when you execute your program you can do this programmatically. Scroll bars will appear if the client area is less than 300 x 250 pixels. We're not quite there yet. We need to take into consideration the scroll position if we have scrolled the window each time we execute `OnPaint`. This is easy. All we need to do is to add the current `AutoScrollPosition` property to the position we draw each graphic. This is actually a negative value. This makes sense since scrolling down or right requires us to draw the graphic above or left of its default position. Above and left corresponds to a negative offset of the coordinate. Here is our complete program:

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;

ScrollEx2
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
```
using System.Windows.Forms;

namespace ScrollEx2
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
            this.AutoScaleMinSize = new Size(300, 250);
        }

        protected override void OnPaint(PaintEventArgs e)
        {
            int X = this.AutoScrollPosition.X;
            int Y = this.AutoScrollPosition.Y;

            Graphics g = e.Graphics;
            g.FillRectangle(Brushes.Gray, 10+X, 10+Y, 75, 23);
            g.FillRectangle(Brushes.Gray, 200+X, 10+Y, 75, 23);
            g.FillRectangle(Brushes.Gray, 10+X, 200+Y, 75, 23);
            g.FillRectangle(Brushes.Gray, 200+X, 200+Y, 75, 23);
        }
    }
}

The `AutoScrollMinSize` property is of type `Size`. It is easiest to set its value by creating a new `Size` structure. There is no need to assign this `Size` structure to an intermediate variable. Figure 3-9 and Figure 3-10 show the output when we run the program and scroll.

Figure 3-9
**DrawString Overloads**

The `DrawString` method has a number of overloads that allow some fairly sophisticated text formatting. The `DrawString` method is a method in the `Graphics` class which is in the `System.Drawing` namespace. Table 1 shows all these overloads.

<table>
<thead>
<tr>
<th>Overload</th>
<th>DrawString Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DrawString (String, Font, Brush, PointF)</td>
</tr>
<tr>
<td>2</td>
<td>DrawString (String, Font, Brush, RectangleF)</td>
</tr>
<tr>
<td>3</td>
<td>DrawString (String, Font, Brush, PointF, StringFormat)</td>
</tr>
<tr>
<td>4</td>
<td>DrawString (String, Font, Brush, RectangleF, StringFormat)</td>
</tr>
<tr>
<td>5</td>
<td>DrawString (String, Font, Brush, Single, Single)</td>
</tr>
<tr>
<td>6</td>
<td>DrawString (String, Font, Brush, Single, Single, StringFormat)</td>
</tr>
</tbody>
</table>

The first three parameters of each overload correspond to the text string to be displayed, the font to use and finally the brush to paint the text. We have already used overload 5 which uses the last two parameters to specify the coordinate of the upper left hand corner of the text. Overload 1 is identical except that the coordinate is specified as a `PointF` structure. A `PointF` is just like a `Point` except that the coordinate is specified in the `float` data type rather than the `int` data type.

One of the first questions most programmers ask is what the policy is regarding the use of the newline character in strings passed to `DrawString`. C and C++ programmers are used to placing `\n` in strings to break the string into two distinct lines. Unfortunately this technique didn't work for Windows programming either using the Win32 API (the operating system interface) or the MFC library's `TextOut` method which is a close match for `DrawString`. .NET was
designed with the programmer in mind and the implementation of `DrawString` does the sensible thing when a `\n` is embedded in the string, it goes to the next line at the horizontal coordinate of the string, but positioned at the proper spacing below the previous line. This is great if we can format the text in advance and position our line breaks appropriately. But what if this isn't possible or convenient? That's where overload 2 comes in handy. Rather than specify the coordinate of the upper left hand corner for the text we specify a `Rectangle` structure that not only specifies the upper left hand corner, but also specifies the width and height of a rectangle to contain the string. The `DrawString` method automatically breaks the text at `whitespace` to fill the rectangle. Here is a simple program that demonstrates the use of this overload. Figure 3-11 show the output if we execute the program.

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

namespace DrawString1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        protected override void OnPaint(PaintEventArgs e)
        {
            string s = "This is a very very very very very very very very very long string";
            Graphics g = e.Graphics;
            g.DrawString(s, Font, Brushes.Black, 10, 10);
            Rectangle rect = new Rectangle(10, 50, 250, 250);
            g.DrawString(s, Font, Brushes.Black, rect);
        }
    }
}
The remaining overloads take a `StringFormat` class. This class allows us to refine the way our text is displayed. The most useful feature is to change the horizontal and/or vertical alignment of the text. There are three options:

<table>
<thead>
<tr>
<th>StringAlignment Enumeration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Center</td>
<td>Text is centered</td>
</tr>
<tr>
<td>Far</td>
<td>Text is left or top aligned</td>
</tr>
<tr>
<td>Near</td>
<td>Text is right or bottom aligned</td>
</tr>
</tbody>
</table>

The `Alignment` and `LineAlignment` properties of the `StringFormat` class are of type `StringAlignment`. Here is a program that centers some text in the client area of our form:

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

namespace DrawString2
{
    public partial class Form1 : Form
    {
        public Form1()
        {
```
InitializeComponent();
this.ResizeRedraw = true;

protected override void OnPaint(PaintEventArgs e)
{
    string s = "This is a very very very very very very very very very very very very very long string";
    Graphics g = e.Graphics;
    StringFormat fmt = new StringFormat();
    fmt.Alignment = StringAlignment.Center;
    fmt.LineAlignment = StringAlignment.Center;
    Rectangle rect = this.ClientRectangle;
    g.DrawString(s, Font, Brushes.Black, rect, fmt);
}

The ClientRectangle property of the Form class is maintained automatically as a Rectangle object that represents the current bounds of the client area. The upper left coordinate is always (0, 0) and the width and height are set appropriately. In this example I have merely used this Rectangle object for the call to the DrawString method. I have also created a StringFormat object with centered alignment both horizontally and vertically. Figure 3-12 shows the output if we run this program and size the form.

![Figure 3-12](image)

The ResizeRedraw Property

You are probably wondering what the statement that sets the property ResizeRedraw to true is needed for. You already know that the Paint event is generated every time Windows needs your application to paint a part of the client area that is not currently on the screen. For efficiency, and to eliminate flicker,
Windows filters out repainting of the area of the form that is not changing. Even though you may be making calls to the methods in the `Graphics` class that appear to output to this area, the actual pixels never reach the display. Normally this is what you want. But will this work for our example? It won't work because we are actually changing the entire contents of the client area because the dimensions of the client area have changed. All the text is reformatted. What we need to do is to repaint the entire client area. To force all output to reach the display we set `ResizeRedraw` to `true`. Each time the form is resized the entire client area is `invalidated` and so everything reaches the display.

Figure 3-13 shows what would happen if we don't set `ResizeRedraw` to `true`. As a rule of thumb you should only set `ResizeRedraw` to `true` if you are changing the layout of the client area if the form is resized.

![Figure 3-13](image)

**Font Basics**

The `Form` class has a property named `Font`. It is of type `Font`. Are you confused? The C# language allows you to use the same identifier for more than one purpose as long as there is no ambiguity. The syntax is used to ascertain exactly what the identifier represents. So the two `Font's` are actually different. In one case it refers to a property and in the other to a class. The `Font` property of the `Form` class is actually inherited from the `Control` class. It's initial value is the same as the `DefaultFont` property of the `Control` class. This value will vary depending on the user's operating system version and the local culture setting of their system.

You might be tempted to just change this property to a font of your choice, such as 20 point Arial. Ooops! What you are actually doing is changing the font for all your controls and not just for the form. The `Font` property is an ambient property. An ambient property is a control property that, if not set, is retrieved from the parent control. You could, of course, set the `Font` property for each of
your controls. This is obviously not very practical. So far we have used the form's
Font property in calls to DrawString. If we just want to paint text in a different font
we can create our own Font object. A thorough discussion of fonts would bore
you at this point when all you want to do is change the font.

First let's actually define what a font is. Contrary to what most people
think, a font is not just the artistic aspect of the characters. That's actually what
we call the typeface. A font consists of a particular set of characters in a specific
typeface and a collection of styles. These styles include italic, bold, underline,
etc. A font may include characters in a limited number of sizes or it may be
scalable. Virtually all fonts nowadays are scalable. The True Type family of fonts
is the most popular for Windows systems and there are hundreds if not
thousands of fonts available. Not all fonts contain the same character set. That's
another important difference.

We can create a new Font object using any one of a large number of
overloaded constructors. Perhaps the most common is:

Font(string familyName, float emSize, FontStyle style)

The string familyName is the exact name of the font you want to use. If the font
doesn't exist on the system the default font will be used, but the size and style
will be applied. The second parameter, emSize, specifies the size of the font in
points. A point is 1/72 of an inch. The last parameter is one of the FontStyle
enumeration members shown in Table 3. These styles may be combined with a
logical or operator. For example, FontStyle.Bold|FontStyle.Italic.

Table 3

<table>
<thead>
<tr>
<th>FontStyle</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bold</td>
<td>Bold text.</td>
</tr>
<tr>
<td>Italic</td>
<td>Italic text.</td>
</tr>
<tr>
<td>Regular</td>
<td>Normal text.</td>
</tr>
<tr>
<td>Strikeout</td>
<td>Text with a line through the middle.</td>
</tr>
<tr>
<td>Underline</td>
<td>Underlined text.</td>
</tr>
</tbody>
</table>

Here is a simple program to create a font and use it for a call to
DrawString. The result is shown in Figure 3-14.

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
```
namespace Font1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        protected override void OnPaint(PaintEventArgs e)
        {
            Font myFont = new Font("Comic Sans MS", 24, FontStyle.Italic);
            e.Graphics.DrawString("Comic Sans MS italic 24 points", myFont, Brushes.Black, 10, 10);
            myFont.Dispose();
        }
    }
}

Calling Dispose

If you notice I have made a call to the Dispose method of the Font class. What does this accomplish? C# cleans up objects created on the heap by invoking the garbage collector periodically. The garbage collector takes care of releasing the storage for all objects that are no longer in scope. If you create an object as a local reference in a method then it goes out of scope when you return from the method. However, there is no guarantee when the garbage collector will be invoked. This is known as non-deterministic destruction. There is a way to invoke the garbage collector at any time, but this is to be avoided as it leads to inefficiency. So why do we care when the garbage collector is run as long as it is run? Normally we don’t. However, some .NET classes encapsulate resources that are created inside the operating system. A font is just one example. A brush is another.
The Dispose method is provided to allow a class to release encapsulated resources. If you call Dispose before returning from the OnPaint method you will ensure that the Windows font resource is released in a timely manner. This may never be an issue for simple programs, but if many Windows resources are not released when we are finished with them then performance will suffer and we may even run out of essential resources.

If you intend to use this font over and over it is a better idea to create the font in the constructor of the Form class. That way you have only one copy that lives for the life of the program instead of creating a new font each time the form is repainted. Here is how to do it:

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

namespace Font2
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
            myFont = new Font("Comic Sans MS", 24, FontStyle.Italic);
        }

        protected override void OnPaint(PaintEventArgs e)
        {
            e.Graphics.DrawString("Comic Sans MS italic 24 points", myFont, Brushes.Black, 10, 10);
        }

        Font myFont;
    }
}
```

**Coordinate Systems**

Thus far we have been using pixels as the graphics unit and device (display) coordinates. The number of pixels displayable on your screen is governed by the current graphics resolution you have set up for your video card, for example, 1024 by 768. Recall that the origin of the coordinate system is the upper left hand corner of the client area of your form and positive x is to the right
and positive $y$ is down. This is not always the most useful coordinate system to use for some applications. Fortunately the .NET FCL includes some nifty features to almost arbitrarily change to the coordinate system of your choice.

Quite often a programmer wants to work with some real unit of measure, for example, inches or millimeters. But what exactly is the correlation between pixels and standard units of measure? If we want to deal with physical devices the actual relationship depends on things like the size of the display, the screen resolution, or in the case of printers the number of dots per inch. Kind of a mess isn’t it? Just consider a program that displays a 100 by 100 pixels box on the screen. How many inches square is this? That would depend on the size of your monitor. The Windows operating system does not take into consideration the size of your monitor. That 100 by 100 pixel square will get larger as your monitor size increases. This would not occur for printers since manufacturers use the actual printed image size. This would mean that a 300 pixel per inch printer would always print our square as 1/3 inch by 1/3 inch regardless of the printer brand. A 600 pixel per inch printer would print this same square as 1/6 inch by 1/6 inch. All this can create some confusion for the programmer. I will discuss printing in Chapter <ref>, For now let’s concentrate on video displays.

Normally we don’t care what the actual size is if we measured things on the screen as long as the size is reasonable for what we are displaying. We expect that a page of text in a word processor will be larger for a larger sized monitor and we never expect an 8 ½ by 11 inch page to be displayed in those exact dimensions. What is important, however, is the size of text. We expect a font with a size of 12 points to be readable and not too big. A 6 point font would be expected to be almost unreadable. Fortunately the Windows operating system makes some good choices and even lets you make some minor changes. Recall that a point is 1/72 of an inch and so a 12 point font should be about 1/6 inch in height. The actual size on the screen depends on the size of the monitor, the screen resolution and the relationship the operating system establishes for pixels per inch. Using the display properties feature you can change the size of the system fonts used or you can change the dots per inch used for all graphics. There are two standard sizes, 96 and 120, as well as the ability to customize. The 96 dpi setting is the default. If you are using a high resolution setting and things appear just a bit too small for you, you can switch to the 120 dpi setting.

The Graphics class has two properties you can use to determine the user’s current setting. Here is a simple program to display these values:

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
```
The `DpiX` and `DpiY` properties are `float` values. These are read only properties and can not be changed. Figure 3-15 shows the result when we run the program with the default system value.

![Figure 3-15](image)

**Page Units**

If you want to display a graphic in such a way that its size did not change with the users setting for dpi you could scale your sizes to the `DpiX` and `DpiY`
properties. This would be a pain in the neck to say the least. Once again the .NET FCL designers have come to our rescue. We can set up a coordinate system that automatically changes with the user’s setting. The `PageUnit` property of the `Graphics` class provides this capability. This property is of type `GraphicsUnit`. Table 4 shows the possible values for this property. Pixel is the default value. Inch, Millimeter, and Point are straightforward fixed units. Millimeters are scaled based on the number of millimeters per inch which is approximately 25.4. Display is the same as Pixels for the screen, but is different for printers. Document is a very precise unit that might be used to format a document that would map nicely to a printer with 300 dpi print resolution. I will discuss the very useful `World` value shortly.

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display</td>
<td>Specifies the unit of measure of the display device. Typically pixels for video displays, and 1/100 inch for printers.</td>
</tr>
<tr>
<td>Document</td>
<td>Specifies the document unit (1/300 inch) as the unit of measure.</td>
</tr>
<tr>
<td>Inch</td>
<td>Specifies the inch as the unit of measure.</td>
</tr>
<tr>
<td>Millimeter</td>
<td>Specifies the millimeter as the unit of measure.</td>
</tr>
<tr>
<td>Pixel</td>
<td>Specifies a device pixel as the unit of measure.</td>
</tr>
<tr>
<td>Point</td>
<td>Specifies a printer's point (1/72 inch) as the unit of measure.</td>
</tr>
<tr>
<td>World</td>
<td>Specifies the world coordinate system unit as the unit of measure.</td>
</tr>
</tbody>
</table>

The following lines of code would draw a 96 pixel by 96 pixel black rectangle if the default dpi setting was used:

```csharp
g.PageUnit = GraphicsUnit.Inch;
g.FillRectangle(Brushes.Black, 1, 1, 1, 1);
```

Figure 3-16 shows the result is as expected.
Now suppose we want to draw a rectangle that is not filled. If we change the two line of code to

```csharp
g.PageUnit = GraphicsUnit.Inch;
g.DrawRectangle(Pens.Black, 1, 1, 1, 1);
```

we get the result of Figure 3-17.
What's wrong here? The size isn't right. The problem is that the default size of a *Pen* is one unit of measurement which in this case is one inch or 96 pixels. This is a pen with a width that is as large as the rectangle itself. How can we fix this? We need to create our own *Pen* object and set the width to an appropriate value. These lines of code set the pen width to 1/100 inch and display the rectangle correctly as shown in Figure 3-18:

```csharp
Pen pen = new Pen(Brushes.Black, 0.01f);
g.PageUnit = GraphicsUnit.Inch;
g.DrawRectangle(pen, 1, 1, 1, 1);
pen.Dispose();
```

![Figure 3-18](image)

We can also set the width of the pen to zero. That results in a width of just one pixel. That might work fine for the display, but sometimes lines just one pixel wide are too faint to see on a printer. A 1/100 inch line on a 600 dpi printer would be 6 pixels rather than 1. You should also note that the smallest width is always one pixel regardless of how small we want to make it.

**Page Scale**

The *Graphics* class has one other property related to the *PageUnit* property, the *PageScale* property. This is a *float* value and is used to scale whatever unit you are using for the *PageUnit* property. For example, you might want a unit that is 1/100 inch. This line of code will accomplish this:
To draw a one inch square rectangle you would use a height and width of 100. This scale factor affects coordinates and the width and height of graphics objects. However, it does not scale fonts. You might expect that if we displayed text using `DrawString` with the scale factor 0.01f that the character height would be extremely small. This is not the case. The size will remain whatever the font size was to begin with. Keep this in mind. There is a way to scale everything. I will get to that next. Here are a few other limitations of the `PageScale` property:

- You can’t use a negative scale factor.
- You can’t change the origin. It is always in the upper left hand corner of the client area.
- You can’t rotate or skew.
- You can set an arbitrary drawing unit by scaling pixels, e.g. 1/10 pixel. However, the fundamental limitation is the resolution of the device itself.

**World Transforms**

When we make calls to the methods in the `Graphics` class we are really using what we call *World Coordinates*. Without a *World Transform* these values are only modified by the *Page Transform* which includes the *PageUnit* and *PageScale* properties. Four world transforms are possible:

1. Translate
2. Scale
3. Rotate
4. Transform matrix

The first three are straightforward. The matrix transform is actually a generalization of the other three except that you have complete control. The use of matrix transforms is best ignored at this point. This advanced feature will probably not be of interest to most programmers unless you are engaged in advanced graphics design.

**The Translate Transform**

The translate transform is extremely useful for a variety of purposes. This transform allows you to place the graphics origin anywhere you want relative to the upper left hand corner of the client area. Two common positions are in the center of the client area and the lower left hand corner. With the origin in the
lower left hand corner we can perform graphics operations with the client area representing the first quadrant of a Cartesian coordinate system. The translate transform has two overloads of which the following is the most commonly used:

void TranslateTransform(float dx, float dy);

The two arguments specify the position of the origin in the client area. It is important to remember that this transform does not change the direction of positive X. Positive X is still down. This means that merely locating the origin in the lower left hand corner does not place us in the first quadrant of a Cartesian coordinate system. In fact we don’t have a Cartesian coordinate system since we would require positive coordinates to be up and right. You will see soon that it is possible to flip the direction of X or Y or both using the scale transform. Let’s place the origin in the lower left hand corner of the client area and draw a 100 by 100 pixel ellipse. Here is how we can do it:

```csharp
namespace WindowsApplication2
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
            ResizeRedraw = true;
        }
        protected override void OnPaint(PaintEventArgs e)
        {
            Graphics g = e.Graphics;
            g.TranslateTransform(0, ClientRectangle.Height);
            g.FillEllipse(Brushes.Black, 0, -100, 100, 100);
        }
    }
}
```

The output is shown in Figure 3-19. The property `ClientRectangle` of the `Form` class provides us with a `Rectangle` object exactly the current size of the client area. The `Height` property specifies the current height. The call to `TranslateTransform` places the origin in the lower right hand corner. Notice that the call to `FillEllipse` places the upper left hand corner of the bounding rectangle
of the ellipse at -100, 100. This is 100 pexels up from the lower left hand corner. Since the ellipse is 100 by 100 it exactly fits in the lower left hand corner regardless of the size of the client area. Setting `ResizeRedraw` to `true` guarantees that we will redraw the ellipse correctly if we change the size of the form.

![Figure 3-19](image)

On page 7 I demonstrated how you can scroll the client area. We accomplished this by adding an X and Y offset to every call to the `Graphics` class. We can simplify this task by using the `TranslateTransform` method. The program’s output is identical. Here is the code:

```csharp
namespace ScrollEx3
{
    using System;
    using System.Collections.Generic;
    using System.ComponentModel;
    using System.Data;
    using System.Drawing;
    using System.Text;
    using System.Windows.Forms;

    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
    }
}
```
this.AutoScaleMinSize = new Size(300, 250);
}

protected override void OnPaint(PaintEventArgs e)
{
    int X = this.AutoScalePosition.X;
    int Y = this.AutoScalePosition.Y;
    Graphics g = e.Graphics;
    g.TranslateTransform(X, Y);
    g.FillRectangle(Brushes.Gray, 10, 10, 75, 23);
    g.FillRectangle(Brushes.Gray, 200, 10, 75, 23);
    g.FillRectangle(Brushes.Gray, 10, 200, 75, 23);
    g.FillRectangle(Brushes.Gray, 200, 200, 75, 23);
}

The Scale Transform

The scale transform allows us to apply a scale to the X and Y coordinates of all calls to the methods of the Graphics class. If the scale is different for X and Y then squares and circles as well as all graphics will not appear as we might expect. The ratio of width to height is known as the aspect ratio. Here is a simple program to draw a circle in the center of the client area that scales to the current size of the client area:

```
namespace Scale1
{
    using System;
    using System.Collections.Generic;
    using System.ComponentModel;
    using System.Data;
    using System.Drawing;
    using System.Text;
    using System.Windows.Forms;

    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
            ResizeRedraw = true;
        }

        protected override void OnPaint(PaintEventArgs e)
        {
            const float width = 100f;
            const float height = 100f;
```
Graphics g = e.Graphics;
float scale = Math.Min(ClientRectangle.Width / 200f, 
    ClientRectangle.Height / 200f);
if (scale == 0f) return;
g.TranslateTransform(ClientRectangle.Width / 2f, 
    ClientRectangle.Height / 2f);
g.ScaleTransform(scale, scale);
g.FillEllipse(Brushes.Black, -width/2f, -height/2f, 
    width, height);
}
}

I have defined a pair of constants to draw a 100 by 100 unit circle. To be sure that all scaling calculations are made using floating point arithmetic I have declared them to be floats. Due to the scaling we will be using units that will not always be integral values. I have also used float constants in all calculations where applicable to further assure that calculations are made properly. Let me give you an example of a common error among programmers. What is the value of x for the following lines of code?

```csharp
int i = 1;
float x = i/2;
```

You might expect it to be 0.5. Unfortunately it won’t be. The compiler first performs an integer division which in this case gives us a zero value and it then converts it to a float. The end result is 0.0. If either or both i and the constant 2 are floats then the calculation will be made properly.

Setting up the scale factor requires determining the relationship between the height and or width of the client area and our circle. If we assume that the diameter of the circle should be \( \frac{1}{2} \) the dimension of the smallest of the height or width of the client area then we can scale the output accordingly. We need to select the smaller of the width or height if we want to preserve the aspect ratio. For a 100 unit diameter circle we need to scale the height or width by dividing by 200. Think of it this way. If the client area is exactly 200 by 200 pixels then the scale factor will be unity and our circle will have a diameter of 100 pixels. If the client area is 400 pixels for the smaller of height or width then the scale factor will be 400/200 or 2. The circle will now be displayed with a 200 pixel diameter and will appear in the same proportions with respect to the client area as for our 200 by 200 pixel client area.

The Math classes Min method is used to select the smaller ratio and this forms the X and Y scale factor. In order to center the circle I have called the TranslateTransform method and set the upper left hand corner of the ellipse’s bounding rectangle to half its width and height. Figures Figure 3-20 and Figure 3-21 Show the result with two different client area sizes. Notice how the smaller dimension of the client area in the second example is used to scale the circles
diameter. There is one remaining line of code to discuss. If the scale factor should become zero by shrinking the form as far as you can, we merely return. The `ScaleTransform` method will throw an exception if any of its arguments are zero.

I want to discuss one more basic fact with regard to the way the `ScaleTransform` works. If we use a negative value then any output is reflected about the relevant axis or both if we use a negative value for both scale factors. Unfortunately this isn’t what we would like in some cases. For example, we may be tempted to use a value of -1 for the Y scale factor to allow the positive Y direction to be up rather than down. Setting the origin in the lower left hand corner of the client area would provide a Cartesian coordinate system using the first quadrant. Unfortunately graphical objects including text will be displayed upside down with respect to their anchor point. For example, a rectangle displayed at coordinate 0,0 will not even be visible since it will be reflected downward and painted below the client area. We can take this into consideration.
for some graphics, but text remains a problem. Here is a simple example if we scale the Y axis to -1 and display some text at 0,-100:

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

namespace Scale2
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        protected override void OnPaint(PaintEventArgs e)
        {
            Graphics g = e.Graphics;
            Font font = new Font("Arial", 24f);
            g.ScaleTransform(1, -1);
            g.DrawString("Upside down!", font, Brushes.Black, 0, -100);
        }
    }
}
```

The output is shown in Figure 3-22. As you can see it is upside down. You should also note that since the origin is still in the upper left hand corner of the client area I had to draw the text at 0,-100 otherwise it will not be displayed. Since the negative scale factor results in positive values going up and negative values going down, a positive Y value would be outside the client area. One final note, the coordinate where the text is drawn remains positioned at the same point in the text which, for this example, is at the bottom left of the character U. That would be the top left if the text had not been reflected vertically. You can reflect graphics vertically, horizontally, or both.
The rotate transform is simple but powerful. Rotating graphics objects can be a very painful operation. Simple graphics such as lines, rectangles, polygons etc. can be rotated in software by applying a simple trigonometric calculation to each and every point and drawing the object with the translated coordinates. Rotating complex graphics objects such as bitmaps and text is extremely difficult without help from the operating system. The RotateTransform method rotates any and all graphics drawn with the methods of the Graphics class. The RotateTransform takes one float argument which is the angle of rotation in degrees. If the angle is greater than 360 degrees the angle of rotation is taken modulo 360. In other words we just keep rotating around and around. A positive value rotates clockwise and a negative value rotates counterclockwise. All rotation is around the origin. Normally it is not useful to rotate with the origin in its default position in the upper left hand corner of the client area. The following program performs these steps:

1. Sets a world coordinate system of 1,000 by 1,000 that always fits into the client area regardless of its size.
2. Translates the origin to the center of the client area.
3. Draws six filled ellipses in a circle by using the rotate transform and one set of arguments to FillEllipse.
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace WindowsApplication2
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
            ResizeRedraw = true;
        }
        protected override void OnPaint(PaintEventArgs e)
        {
            Graphics g = e.Graphics;
            Size cs = ClientSize;
            g.TranslateTransform(cs.Width / 2.0f, cs.Height / 2.0f);
            int scale = Math.Min(cs.Width, cs.Height);
            g.ScaleTransform(scale / 1000.0f, scale / 1000.0f);
            for (int i = 0; i < 6; ++i)
            {
                g.FillEllipse(Brushes.Black, -50, -250, 100, 100);
                g.RotateTransform(60.0f);
            }
        }
    }
}

**ClientSize** is an alternative to **ClientRect** that returns a **Size** object which just holds the height and width of the client area. The first ellipse drawn is 100 by 100 and the upper left hand corner is at -50,-250. This places the center 200 units above the very center of the client area which is 1000 units for the smaller of the height or width. Since we are drawing six circles we place a call to **FillEllipse** and **RotateTransform** inside a simple for loop. We are exploiting a very important feature of the world transforms in that they are **cumulative**. In other words each call to rotate by 60 degrees adds an additional 60 degrees rotation to the previous angle. This might not be as you expected. Figures Figure 3-23 and Figure 3-24 show the results of our program for two different client area sizes. Notice that both scaling and rotation are evident.
It is possible to clear the current value of all the world transforms we have applied. The ResetTransform method accomplishes this. It takes no arguments. Remember that it does not just clear the last transform applied, but all of the transforms. I have left out a number of more advance concepts that you are most likely not interested in at this point. These include a discussion on the affects of applying transforms in different orders and using the more general transform matrix for sophisticated operations such as skewing a graphic.