Forms, Text, and Graphics – A Closer Look

The C# Property

- A property is a way of accessing information contained in a class, such as a field, in a controlled manner.
- This is safer than making the field *public*.
- A property has one or both of a *get* and *set* method.
- The keyword *value* is used to access the value being assigned to the property.

Defining a Property (Example)

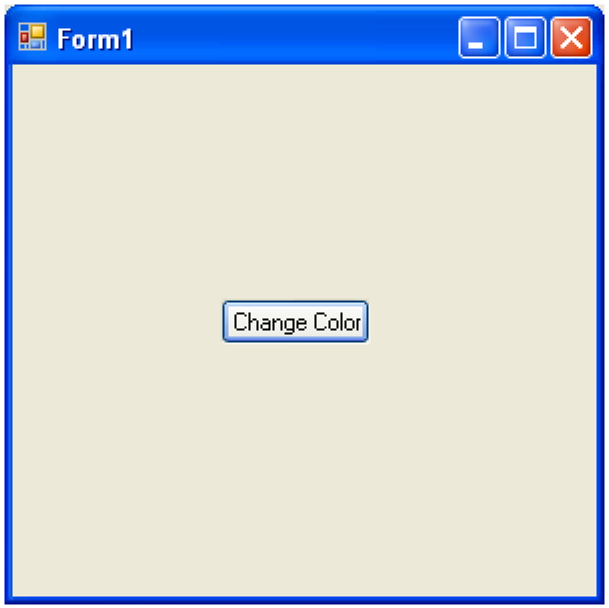
```
public int publicProperty
        {
            get
            {
                return hiddenProperty;
            }
            set
            {
                hiddenProperty = value;
            }
        }
 private int hiddenProperty;
```

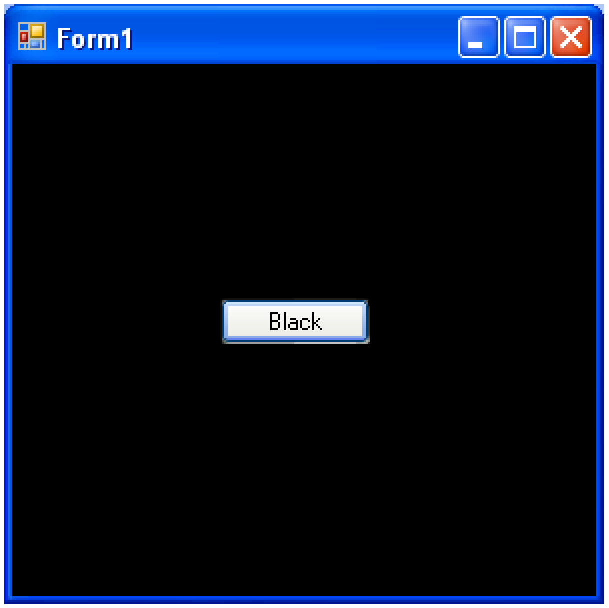
Changing the Background Color of a Form

```
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
    {
        public Form1()
        {
            InitializeComponent();
        }
        private void button1_Click(object sender,
          EventArgs e)
        {
            this.BackColor = Color.Black;
            button1.Text = "Black";
        }
    }
}
```

Form property

Output





Colors

- *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.
- You don’t need to call *Invalidate*.

Overriding the OnPaint Method

- *Form* which is derived from *Control* has an event handler for the *Paint* event, the *OnPaint* method.
- It is declared *virtual* and so it can be overridden in your derived class.
- This is an alternative to using your own event handler and is the recommended technique.
- The *override* keyword is used. This is not the same as C++. *Protected* is also required as the handler in the base class is *protected*.

Overriding the OnPaint Method

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

Good Practice

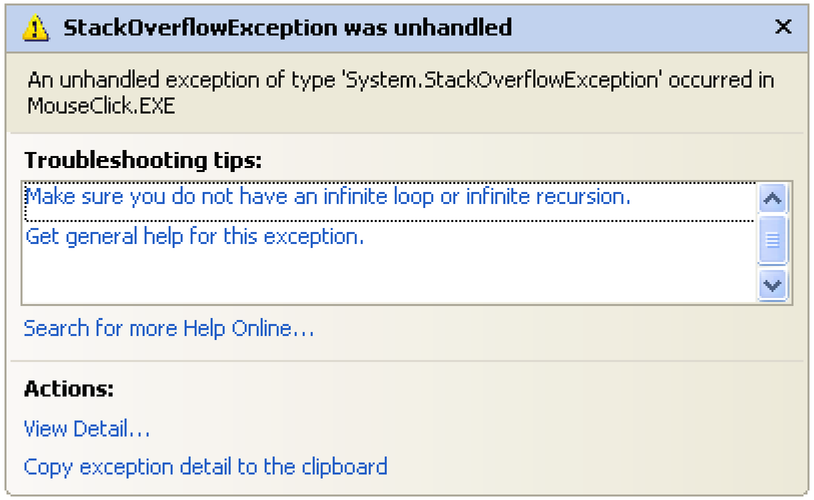
- It is strongly suggested that you call the base class method to make sure that other registered delegates receive the event.
- In other words, if we have other event handlers associated with the *Paint* event we want to make sure that they are called as well.
- Note the required use of the *base* keyword. This is also different than C++.

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

Use base keyword here

Oops!

- If you forget the *base* 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 the next slide 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 2010 is amazing in that it even makes the suggestion that you may have an infinite recursion in your program.



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...

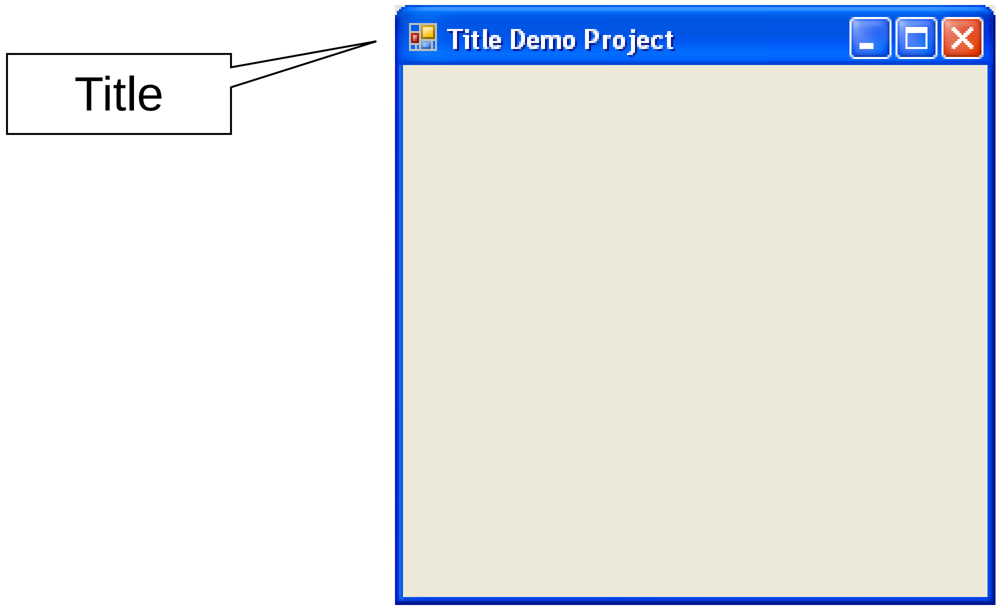
Actions:

View Detail...

Copy exception detail to the clipboard

The Form’s Title Property

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


Title
Title Demo Project

The Form's Position and Size

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 Enumeration

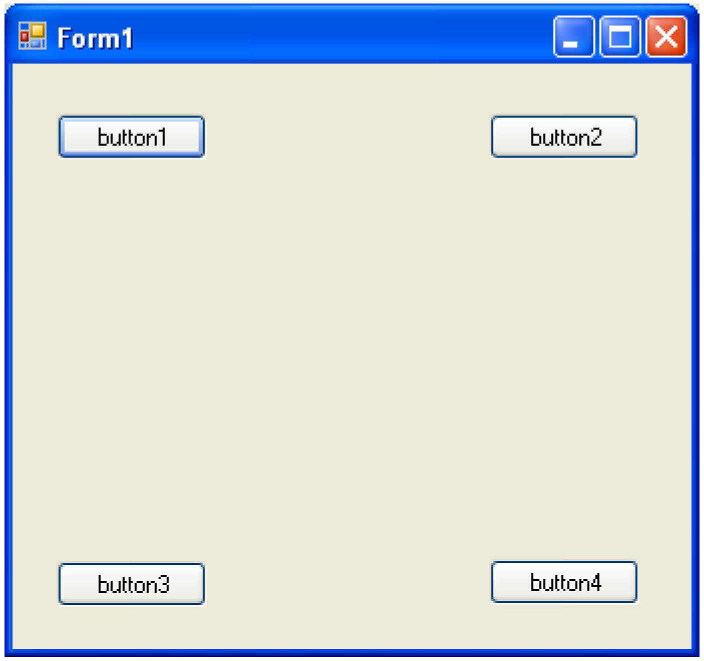
Member	*Action*
CenterParent	Centers the form in the parent form (if exists).
CenterScreen	Centers the form on the screen.
Manual	Location and Size properties are used.
WindowsDefaultBounds	Size and position determined by Windows
WindowsDefaultLocation	Position only is determined by Windows

Also check out the *WindowState* property in the book.

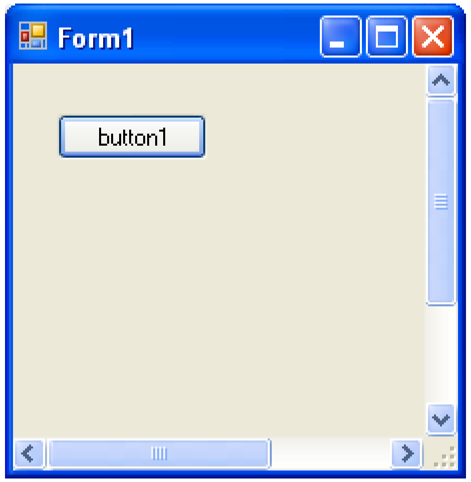
Scrolling a Form

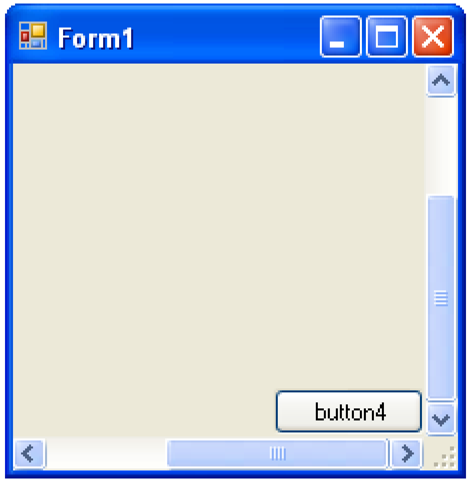
- WIN32 provides a scrolling function but it is not useful for .NET.
- If all the content of our form consists of controls such as buttons we can add scroll bars to a form quite easily. Unfortunately this technique doesn't work if we paint the form using the *Graphics* class.
- To scroll a form with just controls all we need to do is set the form's *Autoscroll* property to *true*.

Example 1



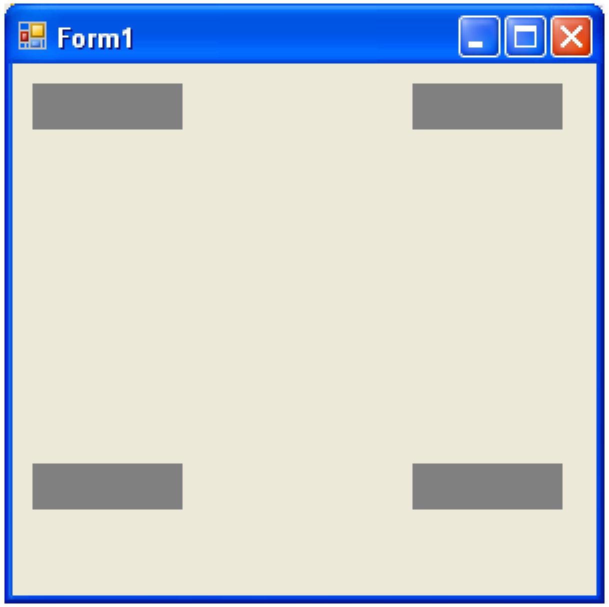
Result





An example that won’t scroll.

```
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);
       g.FillRectangle(Brushes.Gray, 10, 200, 75, 23);
       g.FillRectangle(Brushes.Gray, 200, 200, 75, 23);
  }
```



ScrollNo Example

Doing our own scrolling.

- Set the *AutoScrollMinSize* to the a size that fits ALL of your content.
- Use the *AutoScrollPosition* property to offset the output as you paint.
- *AutoScrollPosition* will have negative values and that is exactly what we want.
- As we scroll down or right, objects that go off the top or left edges of the form are at negative coordinates.

Scroll Example 2

```
//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()
        {
```

Scroll Example 2 – contd.

```
            InitializeComponent();
            AutoScrollMinSize = 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);
        }
    }
}
```

DrawString Overloads

- Both integer and floating point values can be used.
- Rectangles specify the bounds of the text.
- *DrawString* automatically wraps text in the rectangle.

Overload	DrawString Methods
1	DrawString (String, Font, Brush, PointF)
2	DrawString (String, Font, Brush, RectangleF)
3	DrawString (String, Font, Brush, PointF, StringFormat)
4	DrawString (String, Font, Brush, RectangleF, StringFormat)
5	DrawString (String, Font, Brush, Single, Single)
6	DrawString (String, Font, Brush, Single, Single, StringFormat)

What About Newlines?

- Newlines are not recognized in strings drawn using the WIN32 API or the MFC library’s *TextOut* method.
- The .NET desginers had the programmer in mind and *DrawString* accepts newlines.
- Use \n as you would in C or C++.

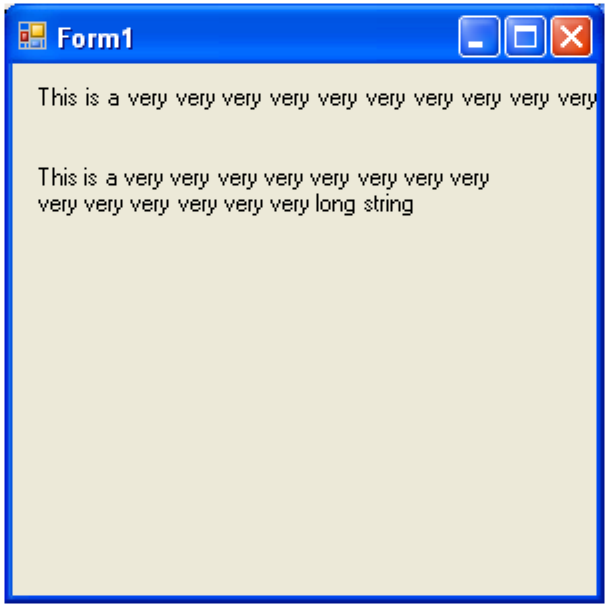
Example 1

```
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()
```

Example – contd.

```
        {
            InitializeComponent();
        }
        protected override void OnPaint(PaintEventArgs e)
        {
            string s = "This is a very very very very ⇦
very 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);
        }
    }
}
```

Output


Form1

```
This is a very very very very very very very very very very

This is a very very very very very very very very
very very very very very very long string
```

The StringFormat Enumeration

StringAllignment Enumeration	*Description*
Center	Text is centered
Near	Text is left or top aligned
Far	Text is right or bottom aligned

StringFormat Example 2

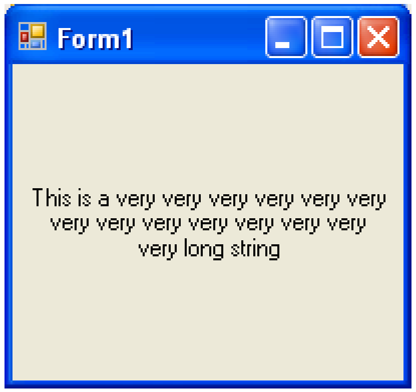
```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
namespace DrawString2
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
```

StringFormat Example – contd.

```
            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 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);
        }
    }
}
```

Example Output

- The *ClientRectangle* property returns the current client area as a *Rectangle* object.

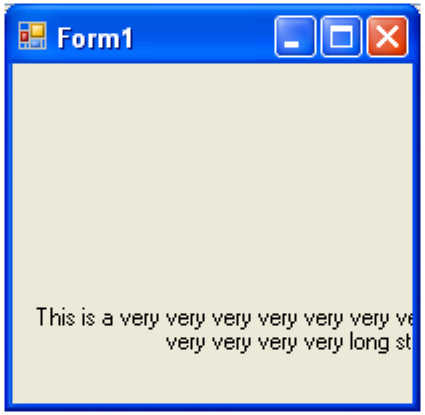


This is a very very very very very very very very very very very very very very long string

The ResizeRedraw Property

- 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.
- Since we are reformatting the *entire* client area we need to ensure that it is completely re-painted.
- Setting *ResizeRedraw* to *true* ensures that the whole client area is repainted if the form is resized.

Without Setting ResizeRedraw to True



This is a very very very very very very ve
very very very very long st

Font Basics

- Font is a class as well as a property in the *Form* class.
- The Font property is an ambient property. An ambient property is a control property that, if not set, is retrieved from the parent control.
- Changing the *Font* property in your form will affect not only the form, but all the controls placed in that form.
- This does not present a major problem since it is very simple to create your own font objects.

Creating a Font

- A font consists of a particular set of characters in a specific typeface and a collection of styles.
- Several constructors are available. One popular is:
```
Font(string familyName, float emSize,
FontStyle style)
```
- *emSize* is the size of the font in points.

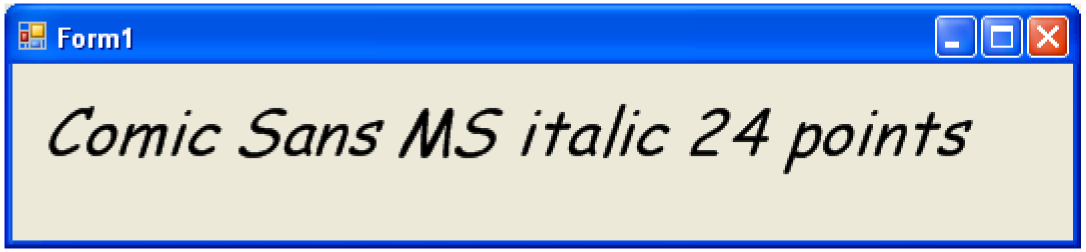
FontStyle Enumeration

FontStyle Enumeration	*Description*
Bold	Bold text.
Italic	Italic text.
Regular	Normal text.
Strikeout	Text with a line through the middle.
Underline	Underlined text.

DrawString Using a Custom Font

```
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();
}
```

Result



Comic Sans MS italic 24 points

Calling Dispose

- Objects that encapsulate Windows objects should be destroyed *deterministically*.
- Using *Dispose* ensures that the underlying Windows object is destroyed without waiting for the garbage collector to run.
- While this is good programming practice it generally doesn't affect programs except in some special cases.
- However, performance may suffer.

Performance Improvement

- Creating the font each time we paint is very inefficient.
- A better method is to create these resources only once for the life of the program.
- Provide a *private* field to reference each object.
- Create the objects in the constructor of your form.

Example

```
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
    {
```

Example – contd.

```
        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);
        }
        private Font myFont;
    }
}
```

.NET Coordinate Systems

- Working in device coordinates (pixels) is not always convenient.
- .NET offers three coordinate systems to work with:
  1. Device
  2. Page
  3. World

Order of Application

$$\begin{tikzcd}
\boxed{\texttt{World}} \arrow[r] & \boxed{\texttt{Page}} \arrow[r] & \boxed{\texttt{Device}}
\end{tikzcd}$$

- The default is that World and Page do not transform coordinates.
- When you call GDI methods you are always using all three transforms.

Pixels and Dots Per Inch

- Windows does not know the size of your monitor and so no actual relationship is possible.
- The basic association is made be the users choice of the font size used by the operating system.
- Normal fonts size is 96 DPI (dots per inch).
- The user can change this with an option under display settings.
- Large fonts might be selected for those with poor vision or when working in high resolution modes on smaller monitors.
- The DpiX and DpiY properties of the Graphics object can be used to determine the current relationship.

Printers

- The default dots per inch depends on the particular printer.
- It may not be square, e.g. horizontal DPI may not equal vertical DPI.
- Device units for printers all default to 0.01 inch rather than pixels.
- Note – this is very close to 96 DPI and so a 100 x 100 box will be relatively the same size on a printer. (e.g., printer looks like 100 DPI to your program)

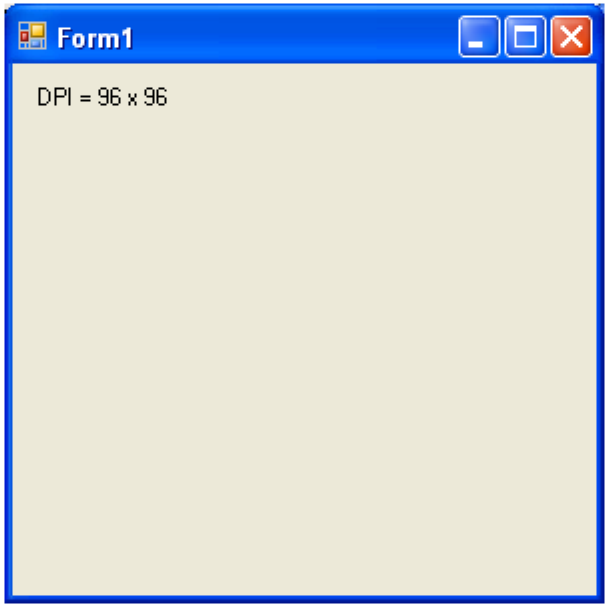
DPI

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

DPI – contd.

```
        {
            InitializeComponent();
        }
        protected override void OnPaint(PaintEventArgs e)
        {
            Graphics g = e.Graphics;
            string s;
            s = "DPI = " + g.DpiX.ToString()
                + " x " + g.DpiY.ToString();
            g.DrawString(s, Font, Brushes.Black, 10, 10);
        }
    }
}
```

Output



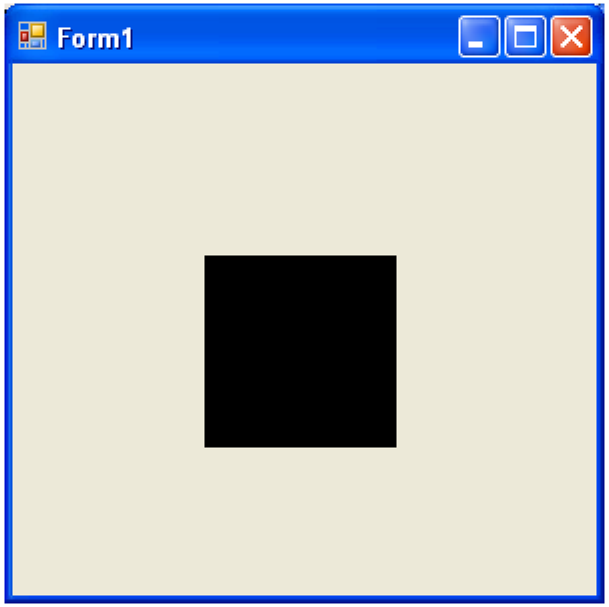
```
DPI = 96 x 96
```

Page Units

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

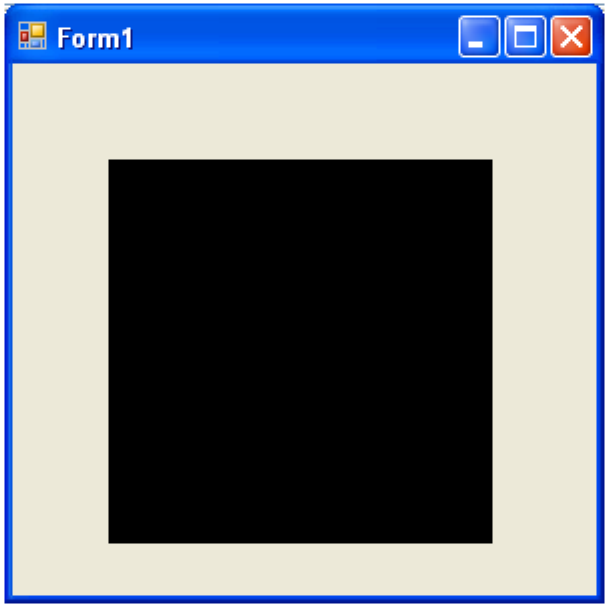
Example

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



The Problem with Pens

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

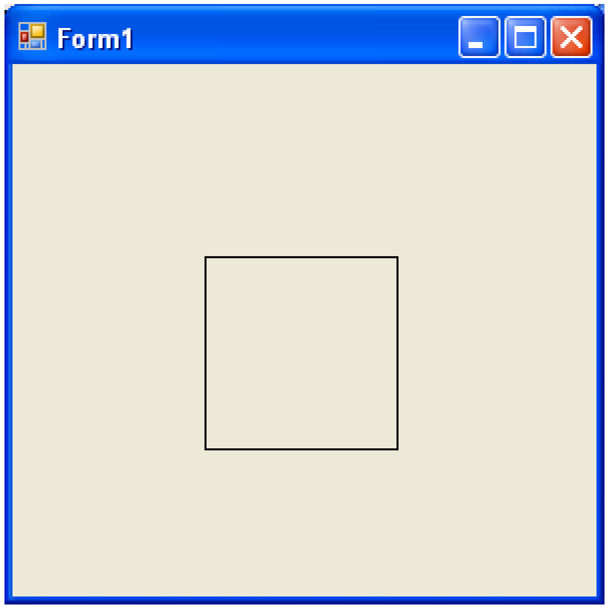


What’s Up?

- The default for a pen is one unit of measure.
- In this case it is one inch.
- Setting the pen width to zero results in a one pixel pen.
- This might be hardly visible on a printer.
- It is better to use a real size.

Better Pen

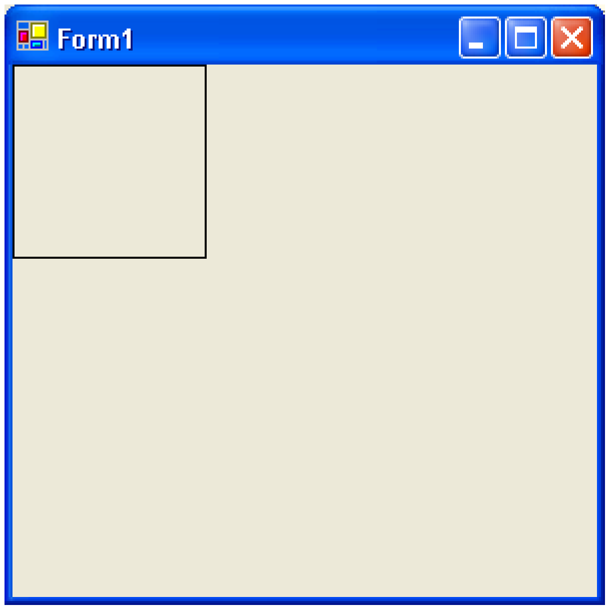
```
Pen pen = new Pen(Brushes.Black, 0.01f);
g.PageUnit = GraphicsUnit.Inch;
g.DrawRectangle(pen, 1, 1, 1, 1);
pen.Dispose();
```



Page Scale

- The PageScale property is used to set a scale factor to the current PageUnit property.
- These two properties are most often used when the pixel is not the optimal unit of measure.
- Example – a CAD program might use a unit of 1/100 millimeter.

```
g.PageUnit=GraphicsUnit.Millimeter;
g.PageScale=0.01f;
g.DrawRectangle(Pens.Black, 0, 0, 2540,
      2540);  //one inch square is drawn
```



Clearly this is not 2540 pixels square.

Page Scale Limitations

- 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 <u>can</u> 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

- Four 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 left up to you to learn. The book does a good job of explaining this.

Order of Application

- The translate, scale, and rotate transforms are <u>not</u> independent.
- You must must apply them in the specific order that makes sense for what you are doing.
- If you scale first, then the unit you use for translation is no longer the page unit (default pixel for screen).
- Successive transforms are cumulative. A new transform is cascaded on top of all other transforms you have made thus far. Beware!

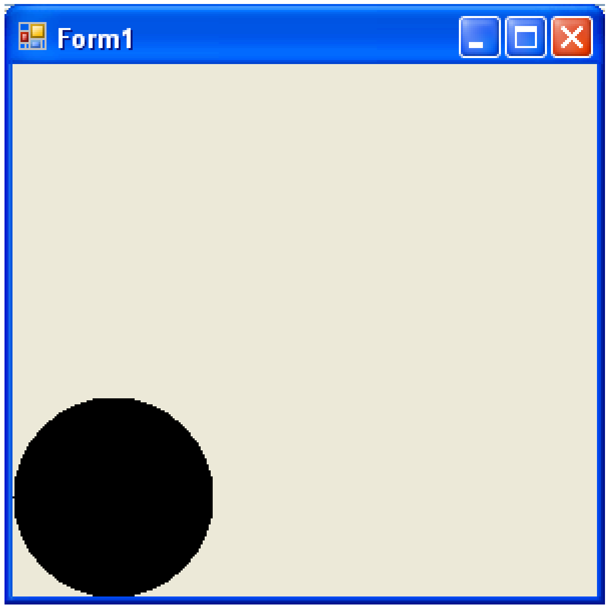
The Translate Transform (translate1)

```
//Translate1
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
    {
```

The Translate Transform – contd.

```
        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);
        }
    }
}
```

Result



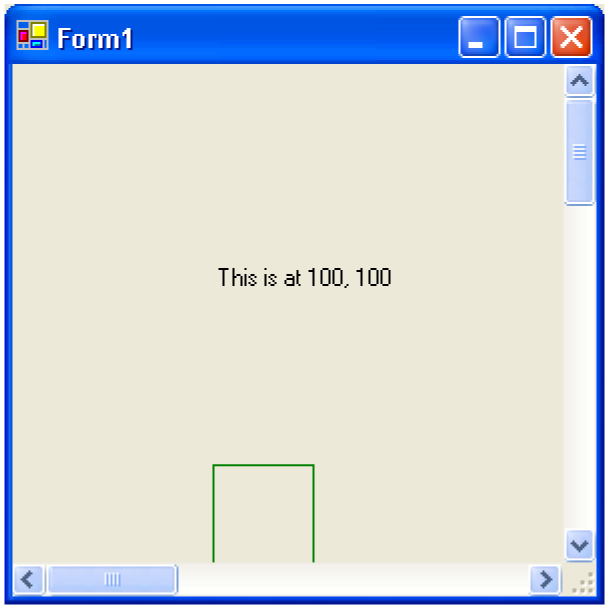
Using Transforms to Scroll

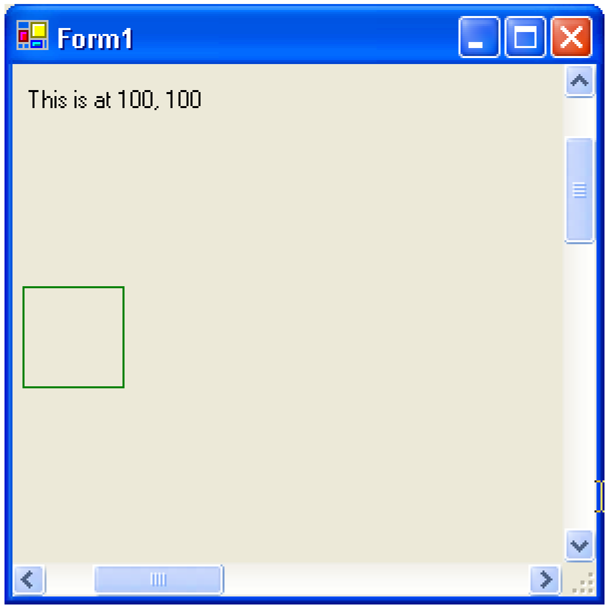
- Rather than adding the ScrollPosition to every call to the GDI it is easier to use the translate transform.
- You must, however, remember that the scroll position is always in pixels.
- The translate transform must be applied before any scale or rotate transforms.

Example

```
    //in the constructor
    this.ResizeRedraw=true;
    this.AutoScroll=true;
    this.AutoScrollMinSize=new Size(1000,1000);


  protected override void OnPaint(PaintEventArgs pea)
  {
    Graphics g=pea.Graphics;
    g.TranslateTransform(this.AutoScrollPosition.X,
        this.AutoScrollPosition.Y);
    g.DrawString("This is at 100, 100", Font,
        Brushes.Black,100, 100);
    g.DrawRectangle(Pens.Green, 100, 200, 50, 50);
  }
```


Form1
This is at 100, 100


Form1
This is at 100, 100

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*.

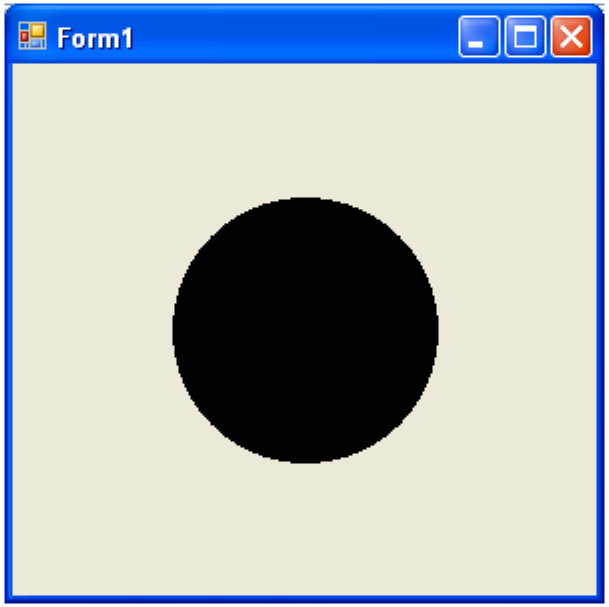
Scale to Client Area (Scale1)

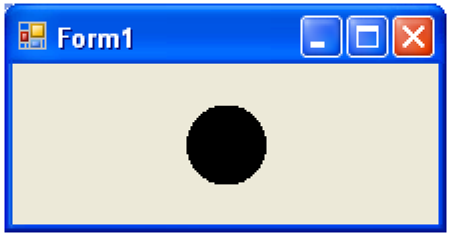
```
//Scale1
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
namespace Scale1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
```

Scale to Client Area – contd.

```
            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);
        }
    }
}
```

Output





Note – the aspect ratio is preserved in this example.

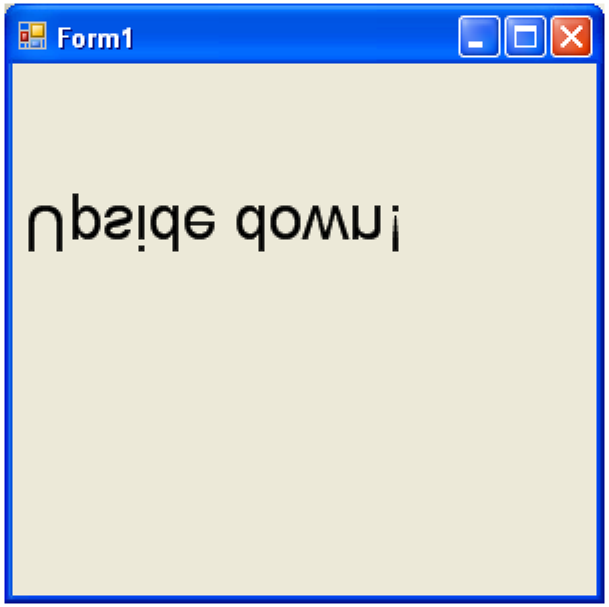
Scale to Reverse the Direction (scale2)

```
/Scale2
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
    {
```

Scale to Reverse the Direction – contd.

```
        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);
        }
    }
}
```

Output



Upside down!

The Rotate Transform

- 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.

A More Interesting Example

- The following program does the following things:
  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.

Rotate Example

```
//Rotate1
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;
```

Rotate Example – contd.

```
        }
        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);
            }
        }
    }
}
```

Result