

Building Rich Client Applications Using C# or Visual Basic .NET



Programming

.NET
Windows
Applications

O'REILLY®

Jesse Liberty & Dan Hurwitz

Drawing and GDI+

The designers of .NET, and especially of Visual Studio .NET, clearly had in mind a model in which you could write sophisticated Windows Applications using only the controls available in the Toolbox. This approach is very successful, and many Windows programmers will never need to go beyond the Toolbox and forms model for building powerful user interfaces.

As discussed elsewhere in this book, the Toolbox includes controls for displaying data (labels, DataGrids, Calendars, listboxes, etc.) as well as for offering the user choices (radio buttons, checkboxes, listboxes, etc.) and for gathering data (text boxes, etc.) In addition, several controls and components manage date and time (Timer, etc.) or the form itself (splitter, etc.).

From time to time, however, you will want to display data in a way that is not possible with just the controls offered in the Toolbox. You might wish to draw on a control, or directly on the form itself, and for that you'll need the tools made available through GDI+ and the Graphics object.

To get an idea of what you can accomplish with these tools, this chapter first covers the basics with simple demonstration programs. It then focuses on a small but complete project, in which you will create an analog clock.

The Drawing Namespace

The System.Drawing namespace includes several classes and structures. The most important of them are summarized, briefly, in Table 10-1.

Table 10-1. Drawing namespace classes and structures

Class	Description
Bitmap	Encapsulates a GDI+ bitmap, i.e., pixel data representing an image.
Brush	Abstract base class. Used to fill the interiors of graphical shapes.
Brushes	Provides static brush definitions for all standard colors.

Table 10-1. Drawing namespace classes and structures (continued)

Class	Description
Color	Structure representing colors, e.g., Color.Green.
Font	Defines a format for text, including font face, and sizeEncapsulates a typeface, size, style, and effects.
FontFamily	Group of type faces with the same basic design.
Graphics	Encapsulates a GDI+ drawing surface.
Icon	Transparent bitmaps used for Windows icons.
Image	Abstract base class common to the Bitmap, Icon, and Metafile classes.
Pen	Defines an object used to draw lines and curves.
Pens	Provides static Pen definitions for all the standard colors.
Point	Structure used to represent an ordered pair of integers. Typically used to specify two-dimensional Cartesian coordinates.
PointF	Same as Point, but uses a floating-point number (float in C#, Single in VB.NET) rather than an integer.
Rectangle	Structure that represents the location and size of a rectangular region.
RectangleF	Same as Rectangle, but uses floating-point values (float in C#, single in VB.NET) rather than integers.
Size	Structure that represents the size of a rectangular region as an order pair (Point) representing width and height.
SizeF	Same as Size, but uses PointF rather than Point.
SystemBrushes	A utility class with 21 static, read-only properties that return objects of type Brush (each of a different color).
SystemPens	A utility class with 15 static, read-only properties that return objects of type Pen (each of a different color).

Arguably the most important class for graphics programming is (surprise!) the Graphics class. The other classes will be described as they are encountered, but before proceeding, let's examine the Graphics class in detail.

The Graphics Class

The Graphics class represents a GDI+ drawing surface. A Graphics object maintains the state of the drawing surface, including the scale and units, as well as the orientation of the drawing surface.

The Graphics class provides many properties. The most commonly used properties are listed in Table 10-2, most of which will be demonstrated later in this chapter.

Table 10-2. Graphics properties

Property	Type	Description
Clip	Region	Read/write. Specifies the area available for drawing.
DpiX / DpiY	Float / single	Read/write. The horizontal and vertical resolution (respectively) of the Graphics object in dots per inch.

Table 10-2. Graphics properties (continued)

Property	Type	Description
PageScale	Float / single	Read/write. The scaling between world units and page units for this Graphics object.
PageUnit	GraphicsUnit	Read/write. The unit of measure for page coordinates. Valid values are members of the GraphicsUnit enumeration, listed in Table 10-3.

The PageScale sets the scaling between the world units and the page units. To understand these concepts, you must first understand coordinates.

Coordinates

The French philosopher René Descartes (1596-1650) is best known today for stating that while he may doubt, he can not doubt that he exists. This is summarized in his oft-quoted statement *Cogito Ergo Sum* (I think, therefore I am).*

Among mathematicians, Descartes may be best known for inventing Analytical Geometry and what are now called Cartesian coordinates. In a classic Cartesian coordinate system, you envision an x axis and a y axis, as shown in Figure 10-1, with the origin (0,0) at the center. The values to the right of the origin and above the origin are positive, and the values to the left and below are negative.

In most graphical programming environments, like in Windows, the coordinate system has its origin at the upper-lefthand corner, rather than in the center, and you count upward to the right and *down*, as shown in Figure 10-2. The coordinates you pass to the various drawing methods of the Graphics class are called *world coordinates*.

Transforms introduced

These world coordinates are transformed into *page coordinates* by *world transformations*. You'll use these world transformations (e.g., TranslateTransform, ScaleTransform, and RotateTransform) later in this chapter to set the center and the orientation of your coordinate system. When drawing a clock face, for example, setting the origin (0,0) to the center of the clock is more convenient.

Page transforms convert page coordinates into device coordinates: pixels relative to the upper-lefthand corner of the client area on your monitor, bitmap, page, etc. The page transforms are the PageUnit and PageScale properties of the Graphics object, which were listed in Table 10-2.

The PageUnit property chooses the unit in which you will make your transformations and scale your drawings. These Units are one of the GraphicsUnits-enumerated values shown in Table 10-3.

* Philosopher joke: Rene Descartes goes into McDonald's and orders a Big Mac. The person behind the counter asks, "You want fries with that?" Descartes replies "I think not," and immediately disappears.

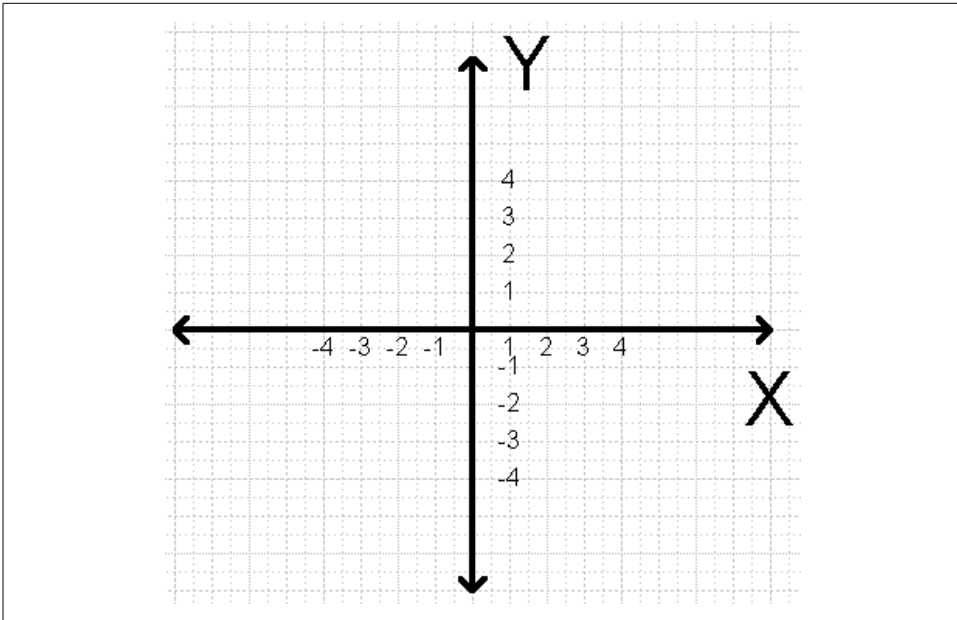


Figure 10-1. Cartesian coordinates

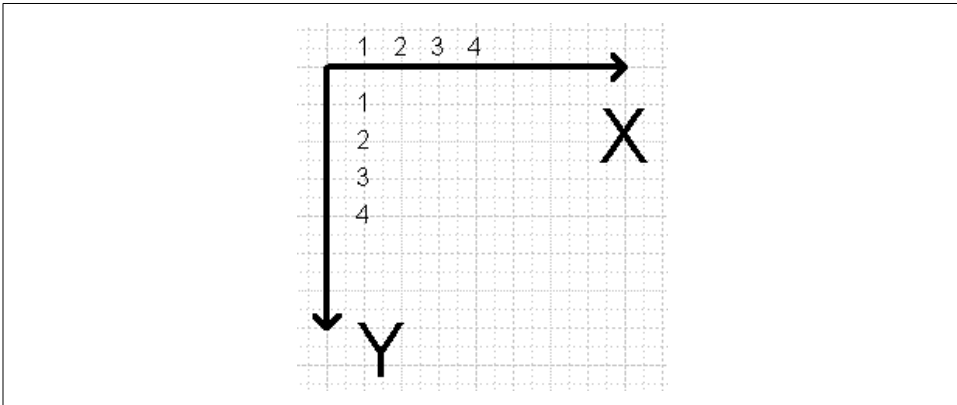


Figure 10-2. World coordinates

Table 10-3. GraphicsUnits enumeration

Enumerated value	Unit of measure
Display	1/75 of an inch
Document	1/300 of an inch
Inch	1 inch
Millimeter	1 millimeter
Pixel	1 Pixel

Table 10-3. GraphicsUnits enumeration (continued)

Enumerated value	Unit of measure
Point	1/72 of an inch
World	World unit

Using the unit described by the PageUnit, you can set the PageScale that specifies the value for scaling between world units and page units for the current Graphics object. You'll see this at work later in this chapter when you'll create a scale of 2000 units by 2,000 units—that is, rather than working in pixels or inches, you'll create an arbitrary unit that is 1/2000 of the width (or height) of your screen.

Graphics Methods

The Graphics class has many methods for drawing to the graphics device. They include methods that begin with *Draw*, shown here:

DrawArc	DrawIcon	DrawPath
DrawBezier	DrawIconUnstretched	DrawPie
DrawBeziers	DrawImage	DrawPolygon
DrawClosedCurve	DrawImageUnscaled	DrawRectangle
DrawCurve	DrawLine	DrawRectangles
DrawEllipse	DrawLines	DrawString

They also include methods that begin with *Fill*, shown here:

FillClosedCurve	FillPie	FillRectangles
FillEllipse	FillPolygon	FillRegion
FillPath	FillRectangle	

(Draw methods draw the outline of the figure; Fill methods fill the interior with the color of the current brush).

The methods you'll use in this chapter, and other commonly used methods, are summarized in Table 10-4.

Table 10-4. Graphics methods

Method	Description
Clear	Clear the drawing area and fill it with the specified color.
DrawString	Draw a string at a particular location using a particular brush and font.
DrawLine	Draw a line connecting two points.
FillEllipse	Fill an ellipse defined by a bounding rectangle.
MeasureString	Return the size of a string when drawn with a specific font within the context of the current Graphics object.
Restore	Restore the state of a Graphics object (see Save, below).

Table 10-4. Graphics methods (continued)

Method	Description
RotateTransform	Apply a rotation to the transformation matrix of a Graphics object.
Save	Save the state of a Graphics object.
ScaleTransform	Apply a scaling operation to the transformation matrix of a Graphics object.
TransformPoints	Transform an array of points from one coordinate space to another, using the current world and page transformations.
TranslateTransform	Apply a translation operation on the transformation matrix.

Create the second hand dot shown in Figure 10-5 using the `FillEllipse` method, and create the clock hands using the `DrawLine` method. Write characters using the `DrawString` method. You can learn more about this process below.

`Save` and `Restore` are two important methods in the `Graphics` class. The `Save` method saves the current state of the `Graphics` class and returns that state in a `GraphicsState` object. You do not need to know anything about the internal mechanism of the `GraphicsState` object. It can be treated as a token that you pass into `Restore`, which restores the saved state of the `Graphics` object. Thus, you might write code that looks like this:

```
C# GraphicsState state = myGraphicsObject.Save();  
    // make transformations here  
    myGraphicsObject.Restore(state);
```

```
VB dim state as GraphicsState = myGraphicsObject.Save()  
    ' make transformations here  
    myGraphicsObject.Restore(state)
```

Using `Save` and `Restore` allows you to change the scale or orientation of your `Graphics` object, and then set it back to the way it was before you transformed it.

The transformation methods are detailed later in this chapter.

Obtaining a Graphics Object

You can obtain a `Graphics` object in many different ways. Two of the ways involve the `Paint` event, which is described in detail in the next section. You can add an event handler to the `Paint` event delegate or override the `OnPaint` event handler method. In either case, the event handler method takes a `PaintEventArgs` parameter, which has a property that returns a `Graphics` object.

```
C# protected override void OnPaint ( PaintEventArgs e )  
    {  
        Graphics g = e.Graphics;
```

```
VB protected override sub OnPaint (e as PaintEventArgs)  
    dim g as Graphics = e.Graphics
```



If you override the `OnPaint` method, you should always finish by chaining up to the base class, forcing it to be invoked. To do so, use the following line of code:

```
C# base.OnPaint(e);
```

```
VB myBase.OnPaint(e)
```

Another way to obtain a `Graphics` object is to call the `CreateGraphics` method on a control or form:

```
C# Graphics g = this.CreateGraphics();
```

```
VB dim g as Graphics = me.CreateGraphics()
```



If you do create a `Graphics` object using `CreateGraphics`, be sure to call the `Dispose` method of the `Graphics` object when you are done with it. Never store a `Graphics` object as a member variable of your own class. Create or obtain the `Graphics` object when you need it and then dispose of it properly. The `C#` `using` keyword can help automate this process for you.

Color Structure

The `System.Drawing` namespace provides a `Color` structure to represent standard system-defined colors. Whenever a color is needed, you can use any of the 141 public static properties of the `Color` structure. These properties include such standbys as `Red`, `Blue`, `Green`, `Gray`, and `Black`; variations on the standards such as `LightBlue`, `LightGreen`, `LightPink`, and `DarkOrange`; and gems such as `LightGoldenRodYellow`, `MediumSlateBlue`, and `PapayaWhip`. The complete list of standard colors is provided in the Appendix.

In addition to the standard colors, 26 members of the `SystemColor` enumeration represent the colors used for various elements of the Windows desktop. They include such items as `ActiveBorder`, `Desktop`, and `WindowFrame`. The complete `SystemColor` enumeration is also shown in the Appendix.

The combination of the standard colors and the `SystemColors` comprise the `KnownColor` enumeration.

The color system is based on the `ARGB` model, where `A` stands for `Alpha`, or the transparency of the color, and `RGB` stands for `Red-Green Blue`. One byte is allocated each for the `Alpha` and the three primary colors. An `Alpha` value of `0` is transparent and `FF` is opaque. Likewise, a zero value for a color (`R`, `G`, or `B`) indicates that none of that color is present, while a value of `FF` indicates that the color is full on.

In addition to the static properties representing the standard colors, several commonly used instance properties of the Color structure are listed in Table 10-5.

Table 10-5. Color structure instance properties

Value	Description
A	Returns byte value of Alpha component
R	Returns byte value of red component
G	Returns byte value of green component
B	Returns byte value of blue component
Name	Returns the name of the color, either user-defined, a known color, or an RGB value

Geometric Structures—Points, Rectangles, and Sizes

The System.Drawing namespace provides several structures for representing a location (Point and PointF), a rectangular area (Rectangle and RectangleF), and a size (Size and SizeF).

All of these structures consist of a pair of read/write ordered pair of numbers. The versions *without* the trailing F in the name take an ordered pair of ordered pairs of integers (four integers in total) and the versions with the trailing F take an ordered pair of ordered pairs of floating-point numbers (float in C# or single in VB.NET). The numbers typically represent pixels. They can also represent other units, such as Inch or Millimeter, hence the need for floating-point as well as integer values.

The integer versions of these structures can be cast to the floating-point version:

```
C#    PointF ptf;  
        Point pt = null;  
        ptf = pt;
```

```
VB    dim ptf as PointF  
        dim pt as Point  
        ptf = pt
```

You cannot cast in the opposite direction, since information may be lost. However, all three integer versions provide static methods (Ceiling, Round, and Truncate) for converting from the floating-point version to the integer version.

In the Point/PointF structure, the first number represents the x, or horizontal coordinate in a two-dimensional Cartesian coordinate system. The second number represents the y, or vertical coordinate.

The Size/SizeF structures are similar to the Point/PointF structures, except that the ordered pair of numbers represent the Width and Height properties of a rectangular region.

The `Rectangle` and `RectangleF` structures represent a rectangular region. Each has two constructors, listed in Table 10-6. The first constructor takes a `Point` (or `PointF`) structure and a `Size` (or `SizeF`) structure. The second constructor takes four numbers, either integers or float/singles. The first two numbers represent the x and y coordinates of the upper-left corner of the rectangle, and the second two numbers represent the width and height of the rectangle.

Table 10-6. *Rectangle and RectangleF constructors*

Rectangle	Point, Size
	Integer, Integer, Integer, Integer
RectangleF	PointF, SizeF
	Integer, Integer, Integer, Integer

The `Rectangle` and `RectangleF` structures also provide several properties for either getting information about the rectangle or setting properties. These properties are listed in Table 10-7.

Table 10-7. *Rectangle and RectangleF properties*

Property	Type	Description
Bottom	Integer Float/single	Read-only. Returns the y coordinate of the bottom edge of the rectangle.
Height	Integer Float/single	Read/write. The height of the rectangle.
IsEmpty	Boolean	Read-only. Returns <code>true</code> if all numeric properties have a value of zero.
Left	Integer Float/single	Read-only. Returns the x coordinate of the left edge of the rectangle.
Location	Point/PointF	Read/write. The point at the upper-left corner of the rectangle.
Right	Integer Float/single	Read-only. Returns the x coordinate of the right edge of the rectangle.
Size	Size/SizeF	Read/write. The size of the rectangle.
Top	Integer Float/single	Read-only. Returns the y coordinate of the top edge of the rectangle.
Width	Integer Float/single	Read/write. The width of the rectangle.
X	Integer Float/single	Read/write. The x coordinate of the upper-left corner of the rectangle.
Y	Integer Float/single	Read/write. The y coordinate of the upper-left corner of the rectangle.

Brush and Brushes

A *Brush* object is used to fill the interior spaces of graphical shapes. The *Brush* class is abstract (*MustInherit*), and five different classes are derived from *Brush*, listed in Table 10-8.

Table 10-8. *Brush-derived classes*

Class	Description
HatchBrush	A rectangular brush with a hatch style from the <i>HatchStyle</i> enumeration, such as <i>BackwardDiagonal</i> , <i>DarkVertical</i> , <i>Divot</i> , and <i>Percent60</i> . The <i>ForegroundColor</i> property gets the color of the lines and the <i>BackgroundColor</i> property gets the color of the space behind the lines.
LinearGradientBrush	A brush with a linear gradient. Properties such as <i>Blend</i> , <i>GammaCorrection</i> , and <i>Transform</i> allow you to change the appearance programatically.
PathGradientBrush	A brush that fills a graphical shape with a gradient. Properties such as <i>Blend</i> , <i>CenterColor</i> , and <i>Transform</i> allow you to change the appearance programatically.
SolidBrush	A brush that fills a graphical shape with a color. Valid colors are members of the <i>Color</i> structure.
TextureBrush	A brush that fills a graphical shape with a texture. The texture can come from an <i>Image</i> object, either a <i>bitmap</i> or a <i>metafile</i> .

For easy access to the colored brushes, a *Brushes* class (note the plural) contains only static (shared), read-only properties of type *Brush* corresponding to each standard color. This *Brushes* class often provides the *Brush* object used in calls to the *DrawString* method.



If you explicitly create a *Brush* (or *Pen*), you must explicitly dispose of it, but you must *not* dispose of any brushes returned to you by *Brushes*, *SystemBrushes*, *Pens*, or *SystemPens*.

Pen and Pens

The *Pen* and *Pens* classes are similar to the *Brush* and *Brushes* classes, except they are used for drawing lines and curves rather than filling graphical shapes. The constructors of the *Pen* class take either a *Brush* object or a *Color* as an argument. Additionally, you can pass in a floating-point number (float in C#, single in VB.NET) that specifies the width of the *Pen* object.

As with the *Brushes* class, there is a *Pens* class (note the plural) for easy access to a *Pen* object of a standard color. The *Pens* class contains static members corresponding to each standard color. The following code snippet demonstrates one way to instantiate and use a *Pen* object.



```
Pen pn = Pens.LimeGreen;  
pn.EndCap = LineCap.ArrowAnchor;  
pn.Width = 20;  
g.DrawLine(pn,0,0,0,50);
```

VB

```
Dim pn as Pen = Pens.LimeGreen
pn.EndCap = LineCap.ArrowAnchor
pn.Width = 20
g.DrawLine(pn, 0, 0, 0, 50)
```

Table 10-9 lists many of the most commonly used properties of the Pen class. Many of them will be demonstrated in the Analog Clock project later in this chapter.

Table 10-9. Pen properties

Property	Type	Description
Alignment	PenAlignment	Read/write. Specifies the alignment for this Pen object relative to a theoretical line. Valid values are members of the PenAlignment enumeration, listed in Table 10-10.
Brush	Brush	Read/write. The Brush object that controls attributes of the pen (e.g., does the pen draw a solid or a pattern?).
Color	Color	Read/write. The color of the Pen. Legal values are members of the Color structure, e.g., Color.DarkGoldenrod.
DashCap	DashCap	Read/write. The cap style used at the beginning and end of the dashes that comprise a dashed line. Valid values are Flat, Round, and Triangle.
DashPattern	float[] / single()	Read/write. An array of numbers specifying the lengths of alternating dashes and spaces.
DashStyle	DashStyle	Read/write. The style used for dashed lines. Valid values are members of the DashStyle enumeration, listed in Table 10-11.
EndCap	LineCap	Read/write. The enumerated arrow type. Valid values are members of the LineCap enumeration, listed in Table 10-12.
StartCap	LineCap	Read/write. The enumerated arrow type. Valid values are members of the LineCap enumeration, listed in Table 10-12.
Width	Float	The thickness of the drawn line.

Table 10-10. PenAlignment enumeration values

Value	Description
Center	Centered over the theoretical line
Inset	Positioned to the inside of the theoretical line
Left	Positioned to the left of the theoretical line
Outset	Positioned to the outside of the theoretical line
Right	Positioned to the right of the theoretical line

Table 10-11. DashStyle enumeration values

Value	Description
Custom	Custom style
Dash	Dashes
DashDot	Repeating pattern of dash-dot
DashDotDot	Repeating pattern of dash-dot-dot

Table 10-11. DashStyle enumeration values (continued)

Value	Description
Dot	Dots
Solid	Solid line

Table 10-12. LineCap enumeration values

Value	Description
AnchorMask	Mask to check if cap is an anchor cap
ArrowAnchor	Arrow-shaped anchor cap
Custom	Custom line cap
DiamondAnchor	Diamond-shaped anchor cap
Flat	Flat line cap
NoAnchor	No anchor
Round	Round line cap
RoundAnchor	Round anchor cap
Square	Square cap
SquareAnchor	Square anchor cap
Triangle	Triangular line cap

Paint Event

All controls have a Paint event, which is raised when the control is about to be drawn. You can add an event handler to the Paint event delegate to dictate how the control is drawn. Using the Paint event, for example, you can draw a string of text directly onto the client area of a form, as shown in Example 10-1 (in C#) and in Example 10-2 (in VB.NET).



For a complete discussion of delegates and events, see Chapter 4.

Example 10-1. Drawing a string with Paint event using C# (PaintDemo.cs)

```
C# using System;
using System.Drawing;
using System.Windows.Forms;

namespace ProgrammingWinApps
{
    public class PaintDemo : Form
    {
        public PaintDemo()
        {
            Text = "Paint Demonstration";
        }
    }
}
```

Example 10-1. Drawing a string with Paint event using C# (PaintDemo.cs) (continued)

```
C#
    Size = new Size(300,200);
    Paint += new PaintEventHandler(PaintHandler);
}

static void Main()
{
    Application.Run(new PaintDemo());
}

private void PaintHandler(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.DrawString("Look Ma, no label!", Font, Brushes.Black, 50, 75);
}
}
```

Example 10-2. Drawing a string with Paint event using VB.NET (PaintDemo.vb)

```
VB
Imports System
Imports System.Drawing
Imports System.Windows.Forms

namespace ProgrammingWinApps
    public class PaintDemo : Inherits Form

        public sub New()
            Text = "Paint Demonstration"
            Size = new Size(300,200)
            AddHandler Paint, AddressOf PaintHandler
        end sub

        public shared sub Main()
            Application.Run(new PaintDemo())
        end sub

        private sub PaintHandler(ByVal sender as object, _
                                ByVal e as PaintEventArgs)
            dim g as Graphics = e.Graphics
            g.DrawString("Look Ma, no label!", Font, Brushes.Black, 50, 75)
        end sub
    end class
end namespace
```

When either program in Example 10-1 or Example 10-2 is compiled and run, you will get the form shown in Figure 10-3.

Both versions of the PaintDemo program start by referencing the required namespaces: System, System.Drawing, and System.Windows.Forms. Inside the constructor of each program, the Text and Size properties are set, and the PaintHandler method is added to the delegate for the Paint event:

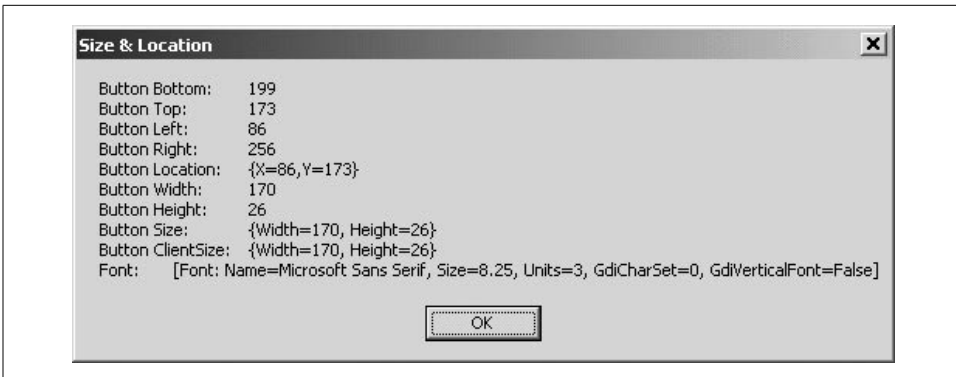


Figure 10-3. Paint event demonstration

```
C#
Text = "Paint Demonstration";
Size = new Size(300,200);
Paint += new PaintEventHandler(PaintHandler);
```

```
VB
Text = "Paint Demonstration"
Size = new Size(300,200)
AddHandler Paint, AddressOf PaintHandler
```

Remember that all of these properties (and events) are implicitly members of the class—i.e., the PaintDemo form. You can make this explicit by using the appropriate lines of code:

```
C#
this.Text = "Paint Demonstration";
this.Size = new Size(300,200);
this.Paint += new PaintEventHandler(PaintHandler);
```

```
VB
Me.Text = "Paint Demonstration"
Me.Size = new Size(300,200)
Me.AddHandler Paint, AddressOf PaintHandler
```

The PaintHandler method is a typical event handler, taking two arguments. The first is of type object and the second is of type PaintEventArgs, which has the two read-only properties listed in Table 10-13.

Table 10-13. PaintEventArgs properties

Property	Type	Description
ClipRectangle	Rectangle	The rectangle to paint
Graphics	Graphics	The Graphics object used to paint

A new Graphics object is instantiated from the PaintEventArgs argument.

```
C# Graphics g = e.Graphics;
```

```
VB dim g as Graphics = e.Graphics
```

Then the DrawString method is invoked to render the desired text string on the form client area. The DrawString method has several overloaded versions, but all of them take at least the string to draw, the font to use, the brush to use (which controls the color and appearance of the characters), and a location:

```
C# g.DrawString("Look Ma, no label!", Font, Brushes.Black, 50, 75);
```

```
VB g.DrawString("Look Ma, no label!", Font, Brushes.Black, 50, 75)
```

In this example, the font used is the current font for the form (it also could have been written as `this.Font` in C# or `me.Font` in VB.NET), the brush is `Black`, and the location of the upper-left corner of the text string is 50 units in from the left edge of the client area and 75 units down from the top.

Overriding the OnPaint method

The programs shown in Examples 10-1 and 10-2 worked by adding an event handler method to the Paint event. You can accomplish the same outcome by overriding the OnPaint event directly just as you might override any virtual method. The Control class defines the OnPaint method as follows:

```
C# protected virtual void OnPaint( PaintEventArgs e );
```

```
VB Overridable Protected Sub OnPaint( ByVal e As PaintEventArgs )
```

Example 10-3 (in C#) and in Example 10-4 (in VB.NET) demonstrate how to override the OnPaint event. The differences from the previous examples are highlighted.

Example 10-3. Overriding the OnPaint event in C# (PaintOverride.cs)

```
C# using System;
using System.Drawing;
using System.Windows.Forms;

namespace ProgrammingWinApps
{
    public class PaintOverride : Form
    {
        public PaintOverride()
        {
            Text = "Paint Demonstration";
        }
    }
}
```

Example 10-3. Overriding the `OnPaint` event in C# (*PaintOverride.cs*) (continued)

```
C#
    Size = new Size(300,200);
}

static void Main()
{
    Application.Run(new PaintOverride());
}

protected override void OnPaint(PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.DrawString("Look Ma, I'm overridden!", Font, Brushes.Black, 50, 75);
    base.OnPaint(e);
}
}
```

Example 10-4. Overriding the `OnPaint` event in VB.NET (*PaintOverride.vb*)

```
VB
imports System
imports System.Drawing
imports System.Windows.Forms

namespace ProgrammingWinApps
    public class PaintOverride : inherits Form

        public sub New()
            Text = "Paint Demonstration"
            Size = new Size(300,200)
        end sub

        public shared sub Main()
            Application.Run(new PaintOverride())
        end sub

        protected overrides sub OnPaint(ByVal e as PaintEventArgs)
            myBase.OnPaint(e)
            dim g as Graphics = e.Graphics
            g.DrawString("Look Ma, I'm overridden!", _
                Font, Brushes.Black, 50, 75)
        end sub
    end class
end namespace
```

In Examples 10-3 and 10-4, no methods are added to any event delegates. Instead, the protected `OnPaint` method is overridden. When overriding an event handler, the only required argument is the event argument—in this case, `PaintEventArgs`.

The last line in the overridden method *chains up* to the base method (invokes the base method) to ensure that all the base class functionality will be implemented and

that any other methods registered with the delegate will be notified. This is done with the line:

```
C# base.OnPaint(e);
```

```
VB myBase.OnPaint(e)
```

The remaining two lines in the overridden method are the same as in the previous examples.

Forcing a paint event—the Invalidate method

You can force a region to redraw by calling the Invalidate method on a control. This method does not actually raise the Paint event, but invalidates the area of the control or a region within the control. Once an area or region has been invalidated, it will be redrawn. This will occur when all current events are finished processing. If you want the redraw to occur immediately, call the Update method after calling the Invalidate method.

The Invalidate method is overloaded with six different versions. The parameters supplied to the method dictate exactly what part of the control will be invalidated. The overloaded versions are listed in Table 10-14.

Table 10-14. Invalidate methods

Method Call	Description
Invalidate()	Invalidates the region of the control
Invalidate(Boolean)	If Boolean is true, invalidates the child controls
Invalidate(Rectangle)	Invalidates the region specified by the Rectangle
Invalidate(Region)	Invalidates the specified Region
Invalidate(Rectangle, Boolean)	Invalidates the region specified by the Rectangle, and if Boolean is true, invalidates the child controls
Invalidate(Region, Boolean)	Invalidates the specified Region, and if Boolean is true, invalidates the child controls

Use of the Invalidate method is shown in Example 10-5 (in C#) and in Example 10-6 (in VB.NET). The lines of code differing from the basic examples in Examples 10-1 and 10-2 are highlighted.

Example 10-5. Invalidate method in C# (PaintInvalidate.cs)

```
C# using System;  
using System.Drawing;  
using System.Windows.Forms;  
  
namespace ProgrammingWinApps
```

Example 10-5. Invalidate method in C# (PaintInvalidate.cs) (continued)

```
C# {
    public class PaintInvalidate : Form
    {
        private Button btn;

        public PaintInvalidate()
        {
            Text = "Paint Invalidate Demonstration";
            Size = new Size(300,200);

            btn = new Button();
            btn.Parent = this;
            btn.Location = new Point(25,25);
            btn.Text = "Update";
            btn.Click += new System.EventHandler(btn_Click);
        }

        static void Main()
        {
            Application.Run(new PaintInvalidate());
        }

        protected override void OnPaint(PaintEventArgs e)
        {
            base.OnPaint(e);
            Graphics g = e.Graphics;
            String str = "Look Ma, I'm overridden!";
            str += "\nThe time is " + DateTime.Now.ToLongTimeString();
            g.DrawString(str, Font, Brushes.Black, 50, 75);
        }

        private void btn_Click(object sender, EventArgs e)
        {
            Invalidate();
        }
    }
}
```

Example 10-6. Invalidate method in VB.NET (PaintInvalidate.vb)

```
VB imports System
imports System.Drawing
imports System.Windows.Forms

namespace ProgrammingWinApps
    public class PaintInvalidate : inherits Form

        private WithEvents btn as Button

        public sub New()
```

Example 10-6. Invalidate method in VB.NET (PaintInvalidate.vb) (continued)

VB

```
Text = "Paint Invalidate Demonstration"
Size = new Size(300,200)

    btn = new Button()
    btn.Parent = me
    btn.Location = new Point(25,25)
    btn.Text = "Update"
end sub

public shared sub Main()
    Application.Run(new PaintInvalidate())
end sub

protected overrides sub OnPaint(ByVal e as PaintEventArgs)
    myBase.OnPaint(e)
    dim g as Graphics = e.Graphics
    dim str as string = "Look Ma, I'm overridden!"
    str += vbCrLf + "The time is " + DateTime.Now.ToLongTimeString()
    g.DrawString(str, Font, Brushes.Black, 50, 75)
end sub

private sub btn_Click(ByVal sender as object, _
    ByVal e as EventArgs) _
    Handles btn.Click
    Invalidate()
end sub
end class
end namespace
```

When either program is compiled and run, you get something similar to that shown in Figure 10-4. Clicking on the button updates the time displayed in the client area of the form.

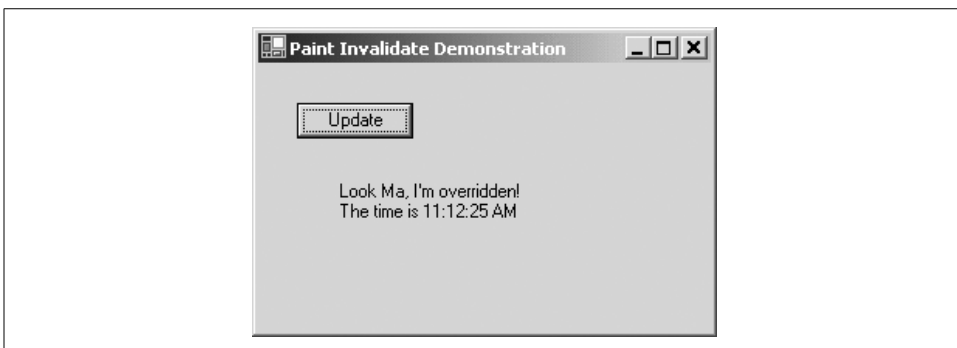


Figure 10-4. Paint Invalidate program

The Invalidate method is called against the Form object. The line:

```
Invalidate()
```

is implicitly the same as:

```
C# this.Invalidate();
```

```
VB me.Invalidate()
```

Another way to force a Form to invalidate itself and cause the Paint event to be raised is to set the `ResizeRedraw` property to `true`. This will cause the object, the Form in this case, to redraw itself every time the object is resized. In fact, if an application is not behaving the way you might expect when resizing, then setting this property to `true` might alleviate the problem.

The Analog Clock Project

To illustrate the use of the Graphics object and GDI+ methods, you'll create a clock face with conventional hour and minute hands and a green dot in lieu of a second hand and a green dot in lieu of a second hand. You will also display the date rotating around the clock face, as shown in Figure 10-5. (If your copy of the book does not display the moving text, you may need to run the program itself, which you will find in Examples 10-11 or 10-12).

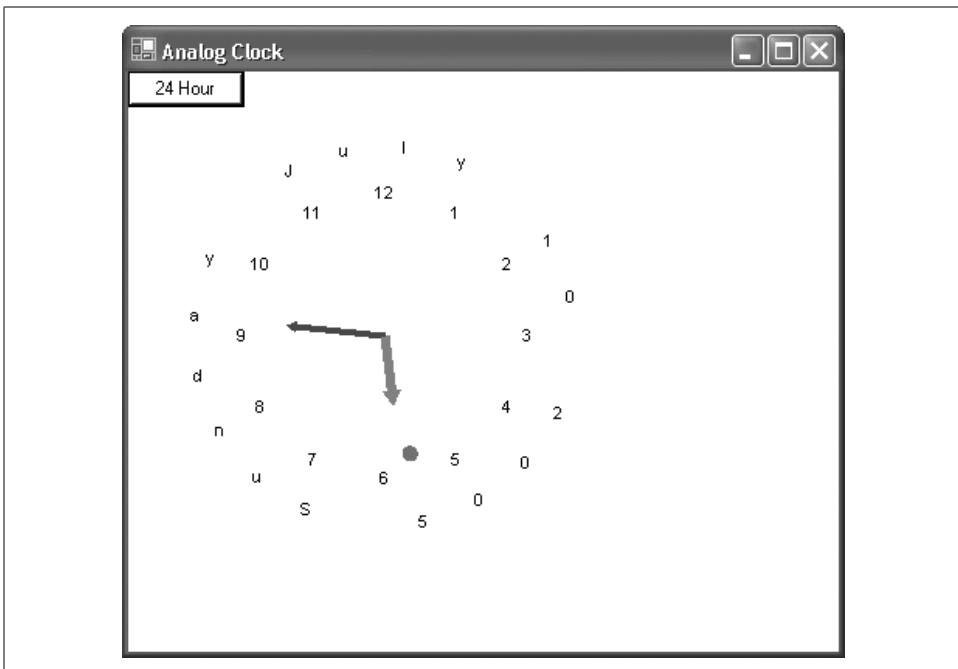


Figure 10-5. Analog Clock (first image)

Notice the button marked “24 Hours” in the upper-lefthand corner. Clicking that button changes the clock to a 24 hour display, as shown in Figure 10-6. Notice that in 24 hour mode, the minute hand maintains its position, but the hour hand must be adjusted.

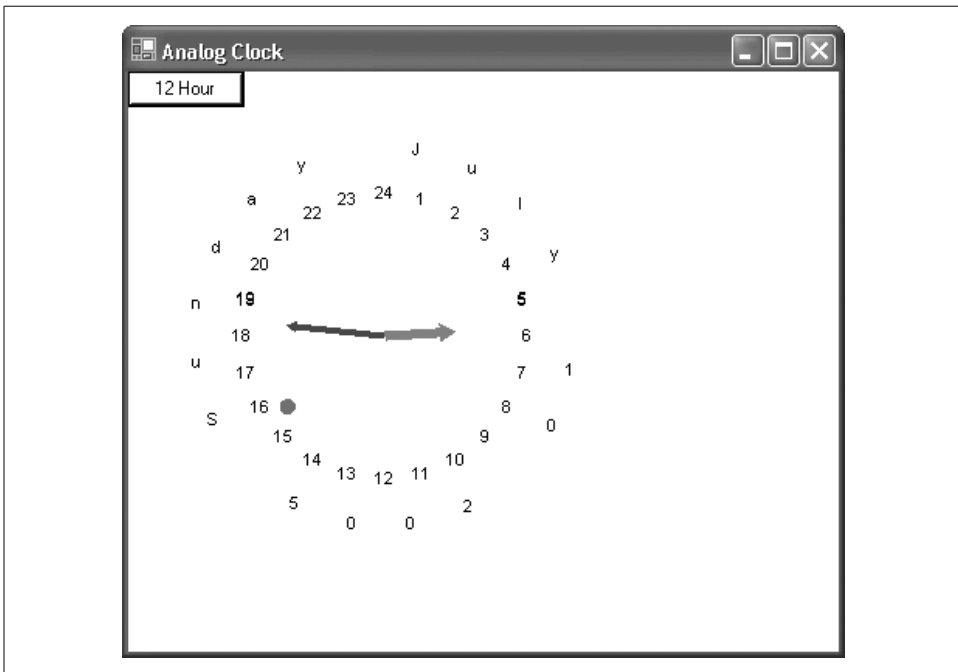


Figure 10-6. Analog Clock 24-hour face

This project presents a number of challenges including those listed next.

- How do you draw a clock face?
- How do you redraw the clock face for 24 hours?
- How do you determine the position of and draw the hands (and the dot for the second hand?)
- How do you draw the date around the outer circumference, and how do you move it so that it rotates around the clock?

As is often the case, each problem has many good solutions, and solving these problems will allow you to explore many details of GDI+ programming.

Drawing the Clock Face

In the first iteration of the clock program, you'll just draw the clock face, as shown in Figure 10-7. The complete source code is shown in Examples 10-7 and 10-8. Detailed analysis follows.

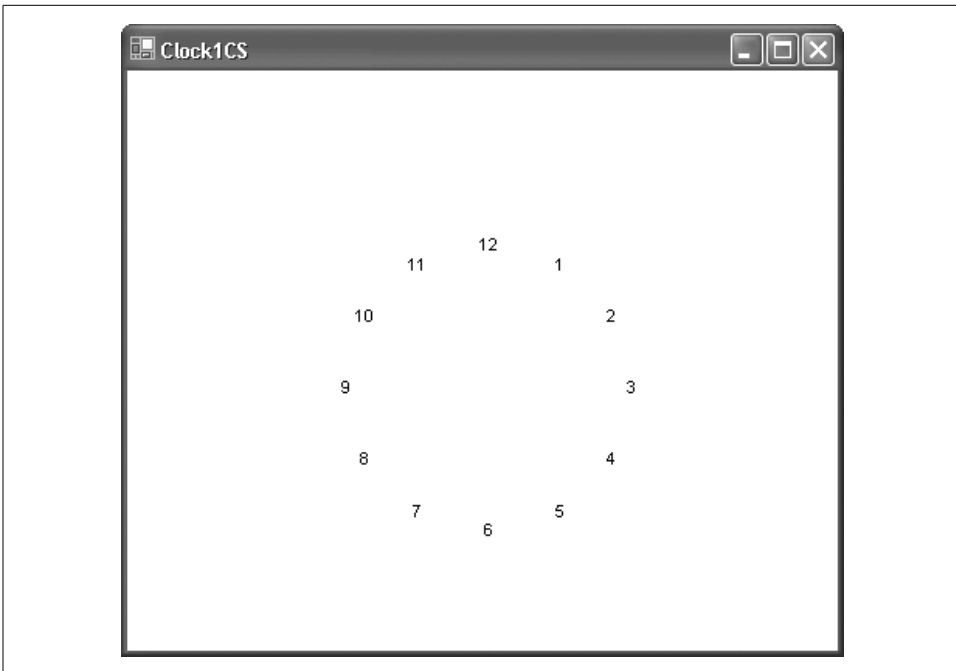


Figure 10-7. Simple clock face

Example 10-7. Drawing the clock face in C#

```
C# using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace Clock1CS
{
    // Summary description for Form1.
    public class Form1 : System.Windows.Forms.Form
    {
        // Required designer variable.
        private System.ComponentModel.Container components = null;

        public Form1()
        {
            // Required for Windows Form Designer support
            InitializeComponent();

            // use the user's choice of colors
            BackColor = SystemColors.Window;
            ForeColor = SystemColors.WindowText;
        }
    }
}
```

Example 10-7. Drawing the clock face in C# (continued)

C#

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

#region Windows Form Designer generated code
protected override void Dispose( bool disposing )
{
    if( disposing )
    {
        if (components != null)
        {
            components.Dispose();
        }
    }
    base.Dispose( disposing );
}

/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
private void InitializeComponent()
{
    //
    // Form1
    //
    this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
    this.ClientSize = new System.Drawing.Size(292, 266);
    this.Name = "Form1";
    this.Text = "Clock1CS";
}
#endregion

[STAThread]
static void Main()
{
    Application.Run(new Form1());
}

private void SetScale(Graphics g)
{
    // if the form is too small, do nothing
    if ( Width == 0 || Height == 0 )
        return;
}
```

Example 10-7. Drawing the clock face in C# (continued)

C#

```
// set the origin at the center
g.TranslateTransform(Width/2, Height/2);

// set inches to the minimum of the width
// or height divided by the dots per inch
float inches = Math.Min(Width / g.DpiX, Height / g.DpiY);

// set the scale to a grid of 2000 by 2000 units
g.ScaleTransform(
    inches * g.DpiX / 2000, inches * g.DpiY / 2000);
}

private void DrawFace(Graphics g)
{
    // numbers are in forecolor except flash number in green
    // as the seconds go by.

    Brush brush = new SolidBrush(ForeColor);
    Font font = new Font("Arial", 40);

    float x, y;

    const int numHours = 12;
    const int deg = 360 / numHours;
    const int FaceRadius = 450;

    // for each of the hours on the clock face
    for (int i = 1; i <= numHours; i++)
    {
        // two ways to do alignment.

        /*
        // 1. figure out size of the string and then
        // offset by half the height and half the width

        // measure the string you're going to draw given
        // the current font
        SizeF stringSize =
            g.MeasureString(i.ToString(), font);

        x = GetCos(i*deg + 90) * FaceRadius;
        x += stringSize.Width / 2;
        y = GetSin(i*deg + 90) * FaceRadius;
        y += stringSize.Height / 2;

        g.DrawString(i.ToString(), font, brush, -x, -y);

        */

        // 2. use a StringFormat object and set
        // its alignment to center
    }
}
```

Example 10-7. Drawing the clock face in C# (continued)

```
C#
    // i = hour  30 degrees = offset per hour
    // +90 to make 12 straight up
    x = GetCos(i*deg + 90) * FaceRadius;
    y = GetSin(i*deg + 90) * FaceRadius;

    StringFormat format = new StringFormat();
    format.Alignment = StringAlignment.Center;
    format.LineAlignment = StringAlignment.Center;

    g.DrawString(
        i.ToString(), font, brush, -x, -y,format);

    } // end for loop
    brush.Dispose();
    font.Dispose();
    } // end drawFace

private static float GetSin(float degAngle)
{
    return (float) Math.Sin(Math.PI * degAngle / 180f);
}

private static float GetCos(float degAngle)
{
    return (float) Math.Cos(Math.PI * degAngle / 180f);
}

} // end class
} // end namespace
```

Example 10-8. Drawing the clock face in VB.NET

```
VB
Imports System
Imports System.Drawing
Imports System.Collections
Imports System.ComponentModel
Imports System.Windows.Forms
Imports System.Data

Namespace ClockFace1

    Public Class Form1
        Inherits System.Windows.Forms.Form

        #Region " Windows Form Designer generated code "
        #End Region

        Protected Overrides Sub OnPaint(ByVal e As PaintEventArgs)
            MyBase.OnPaint(e)
        End Sub
    End Class
End Namespace
```

Example 10-8. Drawing the clock face in VB.NET (continued)

VB

```
Dim g As Graphics = e.Graphics
SetScale(g)
DrawFace(g)
End Sub 'OnPaint

Private Sub SetScale(ByVal g As Graphics)
    ' if the form is too small, do nothing
    If Width = 0 Or Height = 0 Then
        Return
    End If
    ' set the origin at the center
    g.TranslateTransform(Width/2, Height/2)

    ' set inches to the minimum of the width or height divided
    ' by the dots per inch
    Dim inches As Single = _
        Math.Min(Width / g.DpiX, Height / g.DpiX)

    ' set the scale to a grid of 2000 by 2000 units
    g.ScaleTransform(_
        inches * g.DpiX / 2000, inches * g.DpiY / 2000)
End Sub 'SetScale

Private Sub DrawFace(ByVal g As Graphics)
    ' numbers are in forecolor except flash number in green
    ' as the seconds go by.
    Dim myBrush = New SolidBrush(ForeColor)
    Dim greenBrush = New SolidBrush(Color.Green)
    Dim myFont As New Font("Arial", 40)
    Dim x, y As Single

    Const numHours As Integer = 12
    Const deg As Integer = 30
    Const FaceRadius As Integer = 450

    ' for each of the hours on the clock face
    Dim i As Integer
    For i = 1 To numHours

        ' two ways to do alignment.
        ' 1. figure out size of the string and then offset by half
        ' the height and half the width
        ' measure the string you're going to draw given
        ' the current font

        ''Dim stringSize As SizeF = _
            g.MeasureString(i.ToString(), font)
```

Example 10-8. Drawing the clock face in VB.NET (continued)

VB

```
''x = GetCos(i * deg + 90) * FaceRadius
''x += stringSize.Width / 2
''y = GetSin(i * deg + 90) * FaceRadius
''y += stringSize.Height / 2
''g.DrawString(i.ToString(), font, brush, -x, -y)

' 2. use a StringFormat object and set its
' alignment to center
' i = hour 30 degrees = offset per hour
' +90 to make 12 straight up
x = GetCos((i * deg + 90)) * FaceRadius
y = GetSin((i * deg + 90)) * FaceRadius

Dim format As New StringFormat()
format.Alignment = StringAlignment.Center
format.LineAlignment = StringAlignment.Center

g.DrawString(i.ToString(), myFont, myBrush, -x, -y, format)
Next i
End Sub 'DrawFace

Private Shared Function GetSin(ByVal degAngle As Single) As Single
    Return CSng(Math.Sin((Math.PI * degAngle / 180.0F)))
End Function 'GetSin

Private Shared Function GetCos(ByVal degAngle As Single) As Single
    Return CSng(Math.Cos((Math.PI * degAngle / 180.0F)))
End Function 'GetCos
End Class 'Form1
End Namespace
```

Color

When you draw the clock face, you'll need to tell the CLR what color to use for the numbers. You might be tempted to use black, which is perfectly appropriate, but it does raise a problem. As noted in Chapter 9, however, the user may have changed the color scheme to a very dark background (even to black), which would make your clock face invisible.

A better alternative is to set the `BackColor` and `ForeColor` for your form based on the `Window` and `WindowText` colors the user has chosen. You can do so in the constructor for the form:

C#

```
BackColor = SystemColors.Window;
ForeColor = SystemColors.WindowText;
```

You can now set the brush color to the foreground color and feel comfortable with your choice.

OnPaint

Each time the form is created or invalidated, its `OnPaint` method is called. You can override the `OnPaint` method to get a `Graphics` object to work with and paint the control as you wish.

Your override will extract the `Graphics` object from the `PaintEventArgs` object passed in as a parameter. It will then pass that `Graphics` object to two methods: `SetScale` and `DrawFace`, described below:

```
C#    protected override void OnPaint ( PaintEventArgs e )
    {
        Graphics g = e.Graphics;
        SetScale(g);
        DrawFace(g);
        base.OnPaint(e);
    }
```

```
VB    Protected Overrides Sub OnPaint(ByVal e As PaintEventArgs)
        Dim g As Graphics = e.Graphics
        SetScale(g)
        DrawFace(g)
    End Sub 'OnPaint
```

Transforming the coordinates

The job of the `SetScale` method is to make the world transformations to set the origin at the center of the form, and to set the scale to an arbitrary grid of 1,000 units in each of the four directions from the center:

```
C#    private void SetScale(Graphics g)
    {
```

```
VB    Private Sub SetScale(ByVal g As Graphics)
```

Start by making sure that the form has at least some width or height:

```
C#    if ( Width == 0 || Height == 0 )
        return;
```

```
VB    If Width = 0 Or Height = 0 Then
        Return
    End If
```

That done, you are ready to set the origin to the center. To do so, call `TranslateTransform` on the `Graphics` object received as a parameter to the method.

The `TranslateTransform` method is overloaded; the version you'll use takes two floating-point numbers (float in C#, single in VB.NET) as parameters: the x-component of the translation and the y-component. You want to move the origin from the

upper left halfway across the form in the x-direction and halfway down the form in the y-direction.



World translations are implemented with matrices. This mathematical concept is beyond the scope of this book, and you do not need to understand the matrices to use the transformations. For more information, however, please either consult the SDK documentation or look at Charles Petzold's excellent book *Programming Microsoft Windows With C#* (Microsoft Press).

The form inherits two properties from `Control` that you'll use: `Width` and `Height`. Each returns its value in pixels:

```
C# g.TranslateTransform(Width/2, Height/2);
```

The effect is to transform the origin (0,0) to the center both horizontally and vertically.

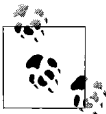
You are now set to transform the scale from its current units (pixels by default) to an arbitrary unit. Don't worry about how large each unit is, but you do want 1,000 units in each direction from the origin, no matter what the screen resolution is. Unfortunately, the size of the units must be equal both horizontally and vertically, so you'll need to choose a size. You will thus compute which size is smaller in inches: the width or the height of the device:

```
C# float inches = Math.Min(Width/g.DpiX, Height/g.DpiX);
```

```
VB Dim inches As Single = Math.Min(Width/g.DpiX, Height/g.DpiX)
```

The variable *inches* now has the smaller of the width or height of the device measured in inches. Multiply that many inches times the dots per inch on the x axis to get the number of dots in the width, and divide by 2,000 to create a unit that is 1/2000th of the width of the form. You'll then do the same for the y axis. If you pass these values to `ScaleTransform`, you'll create an arbitrary scale 2,000 units on the x axis and 2,000 units on the y axis, or 1,000 units in each direction from the center.

```
C# g.ScaleTransform(  
    inches * g.DpiX/2000, inches * g.DpiY/2000);
```



To see this computation for `ScaleTransform` more clearly, you might use interim variables:

```
totalDotsX = inches * g.DpiX;  
numDotsIn2000UnitsX = totalDotsX / 2000;  
  
totalDotsY = inches * g.DpiY;  
numDotsIn2000UnitsY = totalDotsY / 2000;  
  
g.ScaleTransform(numDotsIn2000UnitsX, numDotsIn2000UnitsY);
```

When this method ends, you have the grid you need to draw the clock face. The DrawFace method actually does the work.

World transforms

To draw this clock, write the strings 1 through 12 in the appropriate location. Specify the location as x,y coordinates, and these coordinates must be on the circumference of an imaginary circle.

To compute the x coordinate, take the hour and multiply it by 30, add 90, convert this value from degrees to radians, take the cosine, and then multiply that result by the radius. The formula for the y coordinate is identical, except that you use the sin rather than the cosine:

```
x = GetCos(i*deg + 90) * FaceRadius;
```

To understand why this formula works, see the sidebar “Computing the x,y Coordinates” in this chapter.

Computing the x,y Coordinates

Compute the x coordinate of a point on a circle by multiplying the cosine of the angle by the radius and you compute the y coordinate of a point on a circle by multiplying the sin of the angle by the radius. (see *PreCalculus with Unit Circle Trigonometry* by David Cohn [West Wadsworth]).

These formulae assume that the center of the circle is the origin of your coordinate system, and that the angle is measured counter clockwise from the positive x axis. They also assume that the y axis is positive above the origin and negative below.

A circle is 360 degrees; to evenly space 12 numbers around the face, each number must be 30 degrees from the previous number. The C# Cosine and Sin functions take their parameters in radians, however, not degrees. You'll need to convert degrees to radians using a simple formula: radians equal degrees times pi, divided by 180.

When creating a clock face, it is convenient to measure the degrees offset from the y axis (aligned with 12 o'clock) rather than the x axis, and to increase the angle as you move clockwise (hence the name) rather than the mathematically traditional counter-clockwise. In addition, the coordinate system you'll be using has y values that are negative above the origin, rather than positive.

You solve all three conversions (using the y axis as the zero angle, moving clockwise, and the required coordinate system) by taking advantage of the fact that the cosine of 90 plus an angle is equal to the opposite of the cosine of 90 minus the angle. Thus, to compute 2 o'clock in this system, you compute that 2 is 60 degrees *clockwise* from 12, add 90, and convert the resulting angle (150) to radians and take the cosine of that value. You can then multiply the result times the radius of the circle and you'll get x,y coordinates that match your coordinate system.

Draw each number on the clock face with the overloaded DrawString method of the Graphics object. Table 10-15 lists the overloaded forms of the DrawString method.

Table 10-15. DrawString method overload list (C# and VB.NET)

Method	Description
void DrawString(string, Font, Brush, PointF); sub DrawString(string, Font, Brush, PointF)	Draw the specified string using the specified font and brush at the specified point.
void DrawString(string, Font, Brush, RectangleF); sub DrawString(string, Font, Brush, RectangleF)	Draw the specified string using the specified font and brush in the specified rectangle.
void DrawString(string, Font, Brush, PointF, StringFormat); sub DrawString(string, Font, Brush, PointF, _ StringFormat)	Draw the specified string using the specified font and brush at the specified point using the specified StringFormat.
void DrawString(string, Font, Brush, RectangleF, StringFormat); sub DrawString(string, Font, Brush, RectangleF, _ StringFormat)	Draw the specified string using the specified font and brush in the specified rectangle using the specified StringFormat.
void DrawString(string, Font, Brush, float, float); sub DrawString(string, Font, Brush, float, float)	Draw the specified string using the specified font and brush at the specified x and y coordinates.
void DrawString(string, Font, Brush, float, float, StringFormat); sub DrawString(string, Font, Brush, float, float, _ StringFormat)	Draw the specified string using the specified font and brush at the specified x and y coordinates using the specified StringFormat.

The version of DrawString you'll use in this example will take five parameters:

- The string to draw (the numbers 1 through 12)
- The font to draw in (e.g., Arial 8)
- A brush to determine the color and texture of the text
- The x coordinate of the upper-lefthand corner of the text
- The y coordinate of the upper-lefthand corner of the text

You know you'll need a brush, and you know you want to draw in the foreground color determined by the user, so create an instance of a SolidBrush, passing in the ForeColor property of the form:

```
C#    Brush brush = new SolidBrush(ForeColor);
```

```
VB    Dim brush = New SolidBrush(ForeColor)
```

You also need a Font object. You'll create a font to represent the font face Arial and the size 40. This size will be relative to your new arbitrary scale, so it is arrived at by trial and error:

```
C#    Font font = new Font("Arial", 40);
```

```
VB    Dim font As New Font("Arial", 40)
```

Next, declare two float variables to hold the x and y coordinates that you will compute using the formula discussed earlier (see “Computing the x,y Coordinates”), as well as a few useful constants:

```
C# float x, y;
    const int numHours = 12;
    const int deg = 360 / numHours;
    const int FaceRadius = 450;
```

```
VB Dim x, y As Single
    Const numHours As Integer = 12
    Const deg As Integer = 360 / numHours
    Const FaceRadius As Integer = 450
```

Create the string to draw by creating a for loop:

```
C# for (int i = 1; i <= numHours; i++)
    {
```

```
VB Dim i As Integer
    For i = 1 To numHours
```

Within that loop, draw each number in turn. The first task is to compute the x,y coordinates on the circle:

```
x = GetCos(i*deg + 90) * FaceRadius;
y = GetSin(i*deg + 90) * FaceRadius;
```

The GetCos and GetSin methods convert the degrees to radians:

```
C# private static float GetSin(float degAngle)
    {
        return (float) Math.Sin(Math.PI * degAngle / 180f);
    }

    private static float GetCos(float degAngle)
    {
        return (float) Math.Cos(Math.PI * degAngle / 180f);
    }
```

```
VB Private Shared Function GetSin(ByVal degAngle As Single) As Single
    Return CSng(Math.Sin((Math.PI * degAngle / 180.0F)))
End Function 'GetSin

    Private Shared Function GetCos(ByVal degAngle As Single) As Single
    Return CSng(Math.Cos((Math.PI * degAngle / 180.0F)))
End Function 'GetCos
```

Once you have the coordinates, you are ready to draw the numbers. The problem, however, is that the x,y coordinates you’ve computed will be the location of the upper-lefthand corner of the numbers you draw. This will result in a slightly lopsided clock.

To fix this, center the string around the point determined by your location formula. You can do this in two ways. In the first approach, measure the string, and then subtract half the width and height from the location. Begin by calling the `MeasureString` method on the `Graphics` object, passing in the string (the number you want to display) and the font in which you want to display it:

```
C# SizeF stringSize =  
    g.MeasureString(i.ToString(),font);
```

```
VB Dim stringSize As SizeF = _  
    g.MeasureString(i.ToString(), font)
```

You get back an object of type `SizeF`. `SizeF` is a struct, described earlier, that has two important properties: `Width` and `Height`. You can now compute the location of the object, and then offset the `x` location by half the width and the `y` location by half the height.

```
C# x = GetCos(i*deg + 90) * FaceRadius;  
    x += stringSize.Width / 2;  
    y = GetSin(i*deg + 90) * FaceRadius;  
    y += stringSize.Height / 2;
```

This works perfectly, but .NET is willing to do a lot of the work for you. The trick of the second approach is to call an overloaded version of the `DrawString` method that takes an additional (sixth) parameter: an object of type `StringFormat`:

```
C# StringFormat format = new StringFormat();
```

```
VB Dim format As New StringFormat()
```

You now set the `Alignment` and `LineAlignment` properties of the `StringFormat` object to set the horizontal and vertical alignment of the text you will display. These properties take one of the `StringAlignment` enumerated values: `Center`, `Far`, and `Near`. `Center` will center the text as you'd expect. The `Near` value specifies that the text is aligned near the origin, while the `far` value specifies that the text is displayed far from the origin. In a left-to-right layout, the `near` position is left and the `far` position is right.

```
format.Alignment = StringAlignment.Center;  
format.LineAlignment = StringAlignment.Center;
```

You are now ready to display the string:

```
g.DrawString(i.ToString(), font, brush, -x, -y,format);
```

The `StringFormat` object takes care of aligning your characters, and your clock face is no longer lopsided.

Adding the Hands

Now it's time to add the hour and minute hands to the clock. You will also implement the second "hand" as a ball that will rotate around the circumference of the

clock. To see this work, add a timer to update the time every second. Also add the button that switches between the 24- and 12-hour clock.

The complete source code is provided in Examples 10-9 and 10-10. A detailed analysis follows.

Example 10-9. Clock face 2 in C#

```
C# using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Timers;
using System.Windows.Forms;

namespace Clock2CS
{
    // Summary description for Form1.
    public class Form1 : System.Windows.Forms.Form
    {
        // Required designer variable.
        private System.ComponentModel.Container components = null;

        private int FaceRadius = 450; // size of the clock face
        private bool b24Hours = false; // 24 hour clock face?

        private System.Windows.Forms.Button btnClockFormat;
        private DateTime currentTime; // used in more than one method

        public Form1()
        {
            // Required for Windows Form Designer support
            InitializeComponent();

            // use the user's choice of colors
            BackColor = SystemColors.Window;
            ForeColor = SystemColors.WindowText;

            // update the clock by timer
            System.Timers.Timer timer = new System.Timers.Timer();
            timer.Elapsed += new System.Timers.ElapsedEventHandler(OnTimer);
            timer.Interval = 500;
            timer.Enabled = true;
        }

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

Example 10-9. Clock face 2 in C# (continued)

C#

```
        DrawTime(g,true);    // force an update
    }

    // every time the timer event fires, update the clock
    public void OnTimer(Object source, ElapsedEventArgs e)
    {
        Graphics g = this.CreateGraphics();

        SetScale(g);
        DrawFace(g);
        DrawTime(g,false);
        g.Dispose();
    }

    #region Windows Form Designer generated code
    #endregion

    [STAThread]
    static void Main()
    {
        Application.Run(new Form1());
    }

    private void SetScale(Graphics g)
    {
        // if the form is too small, do nothing
        if ( Width == 0 || Height == 0 )
            return;

        // set the origin at the center
        g.TranslateTransform(Width/2, Height/2);

        // set inches to the minimum of the width
        // or height divided by the dots per inch
        float inches = Math.Min(Width / g.DpiX, Height / g.DpiX);

        // set the scale to a grid of 2000 by 2000 units
        g.ScaleTransform(
            inches * g.DpiX / 2000, inches * g.DpiY / 2000);
    }

    private void DrawFace(Graphics g)
    {
        // numbers are in forecolor except flash number in green
        // as the seconds go by.
        Brush brush = new SolidBrush(ForeColor);
        Font font = new Font("Arial", 40);
        float x, y;

        // new code
        int numHours = b24Hours ? 24 : 12;
```

Example 10-9. Clock face 2 in C# (continued)

C#

```
int deg = 360 / numHours;

// for each of the hours on the clock face
for (int i = 1; i <= numHours; i++)
{
    // i = hour 30 degrees = offset per hour
    // +90 to make 12 straight up
    x = GetCos(i*deg + 90) * FaceRadius;
    y = GetSin(i*deg + 90) * FaceRadius;

    StringFormat format = new StringFormat();
    format.Alignment = StringAlignment.Center;
    format.LineAlignment = StringAlignment.Center;

    g.DrawString(
        i.ToString(), font, brush, -x, -y,format);
} // end for loop
} // end drawFace

private void DrawTime(Graphics g, bool forceDraw)
{
    // length of the hands
    float hourLength = FaceRadius * 0.5f;
    float minuteLength = FaceRadius * 0.7f;
    float secondLength = FaceRadius * 0.9f;

    // set to back color to erase old hands first
    Pen hourPen = new Pen(BackColor);
    Pen minutePen = new Pen(BackColor);
    Pen secondPen = new Pen(BackColor);

    // set the arrow heads
    hourPen.EndCap = LineCap.ArrowAnchor;
    minutePen.EndCap = LineCap.ArrowAnchor;

    // hour hand is thicker
    hourPen.Width = 30;
    minutePen.Width = 20;

    // second hand
    Brush secondBrush = new SolidBrush(BackColor);
    const int EllipseSize = 50;

    GraphicsState state; // to protect and to serve

    // Step 1. Delete the old time

    // delete the old second hand
    // figure out how far around to rotate to draw the second hand
```

Example 10-9. Clock face 2 in C# (continued)

C#

```
// save the current state, rotate, draw and then restore the
// state
float rotation = GetSecondRotation();
state = g.Save();
g.RotateTransform(rotation);
g.FillEllipse(
    secondBrush,
    -(EllipseSize/2),
    -secondLength,
    EllipseSize,
    EllipseSize);
g.Restore(state);

DateTime newTime = DateTime.Now;
bool newMin = false; // has the minute changed?

// if the minute has changed, set the flag
if ( newTime.Minute != currentTime.Minute )
    newMin = true;

// if the minute has changed or you must draw anyway then you
// must first delete the old minute and hour hand
if ( newMin || forceDraw )
{
    // figure out how far around to rotate to draw the minute hand
    // save the current state, rotate, draw and then
    // restore the state
    rotation = GetMinuteRotation();
    state = g.Save();
    g.RotateTransform(rotation);
    g.DrawLine(minutePen,0,0,0,-minuteLength);
    g.Restore(state);

    // figure out how far around to rotate to draw the hour hand
    // save the current state, rotate, draw and then
    // restore the state
    rotation = GetHourRotation();
    state = g.Save();
    g.RotateTransform(rotation);
    g.DrawLine(hourPen,0,0,0,-hourLength);
    g.Restore(state);
}

// step 2 - draw the new time
currentTime = newTime;

hourPen.Color = Color.Red;
minutePen.Color = Color.Blue;
secondPen.Color = Color.Green;
secondBrush = new SolidBrush(Color.Green);
```

Example 10-9. Clock face 2 in C# (continued)

C#

```
// draw the new second hand
// figure out how far around to rotate to draw the second hand
// save the current state, rotate, draw and then restore the
// state
state = g.Save();
rotation = GetSecondRotation();
g.RotateTransform(rotation);
g.FillEllipse(
    secondBrush,
    -(EllipseSize/2),
    -secondLength,
    EllipseSize,
    EllipseSize);
g.Restore(state);

// if the minute has changed or you must draw anyway then you
// must draw the new minute and hour hand
if ( newMin || forceDraw )
{
    // figure out how far around to rotate to draw the minute hand
    // save the current state, rotate, draw and then
    // restore the state
    state = g.Save();
    rotation = GetMinuteRotation();
    g.RotateTransform(rotation);
    g.DrawLine(minutePen,0,0,0,-minuteLength);
    g.Restore(state);

    // figure out how far around to rotate to draw the hour hand
    // save the current state, rotate, draw and then
    // restore the state
    state = g.Save();
    rotation = GetHourRotation();
    g.RotateTransform(rotation);
    g.DrawLine(hourPen,0,0,0,-hourLength);
    g.Restore(state);
}
}

// determine the rotation to draw the hour hand
private float GetHourRotation()
{
    // degrees depend on 24 vs. 12 hour clock
    float deg = b24Hours ? 15 : 30;
    float numHours = b24Hours ? 24 : 12;
    return( 360f * currentTime.Hour / numHours +
        deg * currentTime.Minute / 60f);
}

private float GetMinuteRotation()
{
```

Example 10-9. Clock face 2 in C# (continued)

```
C#
    return( 360f * currentTime.Minute / 60f );
}

private float GetSecondRotation()
{
    return(360f * currentTime.Second / 60f);
}

private static float GetSin(float degAngle)
{
    return (float) Math.Sin(Math.PI * degAngle / 180f);
}

private static float GetCos(float degAngle)
{
    return (float) Math.Cos(Math.PI * degAngle / 180f);
}

private void btnClockFormat_Click(object sender, System.EventArgs e)
{
    btnClockFormat.Text = b24Hours ? "24 Hour" : "12 Hour";
    b24Hours = ! b24Hours;
    this.Invalidate();
}

} // end class
} // end namespace
```

Example 10-10. Clock face 2 in VB.NET

```
VB
Imports System
Imports System.Collections
Imports System.ComponentModel
Imports System.Data
Imports System.Drawing
Imports System.Drawing.Drawing2D
Imports System.Timers
Imports System.Windows.Forms

Namespace ClockFace1

    Public Class Form1
        Inherits System.Windows.Forms.Form

        Private FaceRadius As Integer = 450 ' size of the clock face
        Private b24Hours As Boolean = False ' 24 hour clock face?
        Private currentTime As DateTime
        Private WithEvents btnClockFormat as Button
    End Class
End Namespace
```

Example 10-10. Clock face 2 in VB.NET (continued)

VB

```
Public Sub New()  
    MyBase.New()  
  
    'This call is required by the Windows Form Designer.  
    InitializeComponent()  
    ' use the user's choice of colors  
    BackColor = SystemColors.Window  
    ForeColor = SystemColors.WindowText  
  
    ' redraw when resized  
    Me.ResizeRedraw = True  
  
    ' update the clock by timer  
    Dim timer As New System.Timers.Timer()  
    AddHandler timer.Elapsed, AddressOf OnTimer  
    timer.Interval = 500  
    timer.Enabled = True  
End Sub  
  
' every time the timer event fires, update the clock  
Public Sub OnTimer( _  
    ByVal source As Object, ByVal e As ElapsedEventArgs)  
    Dim g As Graphics = Me.CreateGraphics()  
  
    SetScale(g)  
    DrawFace(g)  
    DrawTime(g, False)  
    g.Dispose()  
End Sub 'OnTimer  
  
#Region " Windows Form Designer generated code "  
#End Region  
  
Protected Overrides Sub OnPaint(ByVal e As PaintEventArgs)  
    MyBase.OnPaint(e)  
    Dim g As Graphics = e.Graphics  
    SetScale(g)  
    DrawFace(g)  
    DrawTime(g, True) ' force an update  
End Sub 'OnPaint  
  
Private Sub SetScale(ByVal g As Graphics)  
    ' if the form is too small, do nothing  
    If Width = 0 Or Height = 0 Then  
        Return  
    End If  
    ' set the origin at the center  
    g.TranslateTransform(Width / 2, Height / 2)  
  
    ' set inches to the minimum of the width or height divided  
    ' by the dots per inch
```

Example 10-10. Clock face 2 in VB.NET (continued)

VB

```
Dim inches As Single = _
    Math.Min(Width / g.DpiX, Height / g.DpiY)

' set the scale to a grid of 2000 by 2000 units
g.ScaleTransform(inches * g.DpiX / 2000, _
    inches * g.DpiY / 2000)
End Sub 'SetScale

Private Sub DrawFace(ByVal g As Graphics)
    ' numbers are in forecolor except flash number in green
    ' as the seconds go by.
    Dim brush = New SolidBrush(ForeColor)
    Dim font As New Font("Arial", 40)
    Dim x, y As Single

    Dim numHours As Integer
    If b24Hours Then
        numHours = 24
    Else
        numHours = 12
    End If
    Dim deg As Integer = 360 / numHours
    Const FaceRadius As Integer = 450

    ' for each of the hours on the clock face
    Dim i As Integer
    For i = 1 To numHours
        ' i = hour 30 degrees = offset per hour
        ' +90 to make 12 straight up
        x = GetCos((i * deg + 90)) * FaceRadius
        y = GetSin((i * deg + 90)) * FaceRadius

        Dim format As New StringFormat()
        format.Alignment = StringAlignment.Center
        format.LineAlignment = StringAlignment.Center

        g.DrawString(i.ToString(), font, brush, -x, -y, format)
    Next i
End Sub 'DrawFace

Private Sub DrawTime( _
    ByVal g As Graphics, ByVal forceDraw As Boolean)

    ' length of the hands
    Dim hourLength As Single = FaceRadius * 0.5F
    Dim minuteLength As Single = FaceRadius * 0.7F
    Dim secondLength As Single = FaceRadius * 0.9F

    ' set to back color to erase old hands first
    Dim hourPen As New Pen(BackColor)
```

Example 10-10. Clock face 2 in VB.NET (continued)

VB

```
Dim minutePen As New Pen(BackColor)
Dim secondPen As New Pen(BackColor)

' set the arrow heads
hourPen.EndCap = LineCap.ArrowAnchor
minutePen.EndCap = LineCap.ArrowAnchor

' hour hand is thicker
hourPen.Width = 30
minutePen.Width = 20

' second hand is in green
Dim secondBrush As New SolidBrush(BackColor)
Const EllipseSize As Single = 50

Dim rotation As Single ' how far around the circle?
Dim state As GraphicsState ' to to protect and to serve
Dim newTime As DateTime = DateTime.Now
Dim newMin As Boolean = False ' has the minute changed?
' if the minute has changed, set the flag
If newTime.Minute <> currentTime.Minute Then
    newMin = True
End If
' 1 - delete the old time
' delete the old second hand
' figure out how far around to rotate to draw the second hand
' save the current state, rotate, draw and then
' restore the state
rotation = GetSecondRotation()
state = g.Save()
g.RotateTransform(rotation)
g.FillEllipse( _
    secondBrush, _
    -(EllipseSize / 2), _
    -secondLength, _
    EllipseSize, _
    EllipseSize)
g.Restore(state)

' if the minute has changed or you must draw anyway then you
' must first delete the old minute and hour hand
If newMin Or forceDraw Then

    ' how far around to rotate to draw the minute hand
    ' save the current state, rotate, draw and then
    ' restore the state
    rotation = GetMinuteRotation()
    state = g.Save()
    g.RotateTransform(rotation)
    g.DrawLine(minutePen, 0, 0, 0, -minuteLength)
    g.Restore(state)
```

Example 10-10. Clock face 2 in VB.NET (continued)

VB

```
' figure out how far around to rotate to draw the
' hour hand save the current state, rotate, draw and then
' restore the state
rotation = GetHourRotation()
state = g.Save()
g.RotateTransform(rotation)
g.DrawLine(hourPen, 0, 0, 0, -hourLength)
g.Restore(state)
End If

' step 2 - draw the new time
currentTime = newTime

hourPen.Color = Color.Red
minutePen.Color = Color.Blue
secondPen.Color = Color.Green
secondBrush = New SolidBrush(Color.Green)

' draw the new second hand
' figure out how far around to rotate to draw the second hand
' save the current state, rotate, draw and then
' restore the state
state = g.Save()
rotation = GetSecondRotation()
g.RotateTransform(rotation)
g.FillEllipse( _
    secondBrush, _
    -(EllipseSize / 2), _
    -secondLength, _
    EllipseSize, _
    EllipseSize)
g.Restore(state)

' if the minute has changed or you must draw anyway then you
' must draw the new minute and hour hand
If newMin Or forceDraw Then

    ' how far around to rotate to draw the minute hand
    ' save the current state, rotate, draw and then
    ' restore the state
    state = g.Save()
    rotation = GetMinuteRotation()
    g.RotateTransform(rotation)
    g.DrawLine(minutePen, 0, 0, 0, -minuteLength)
    g.Restore(state)

    ' figure out how far around to rotate to draw the hour hand
    ' save the current state, rotate, draw and then
    ' restore the state
    state = g.Save()
    rotation = GetHourRotation()
    g.RotateTransform(rotation)
```

Example 10-10. Clock face 2 in VB.NET (continued)

VB

```
        g.DrawLine(hourPen, 0, 0, 0, -hourLength)
        g.Restore(state)
    End If
End Sub 'DrawTime

' determine the rotation to draw the hour hand
Private Function GetHourRotation() As Single
    ' degrees depend on 24 vs. 12 hour clock
    Dim deg As Single
    Dim numHours As Single
    If b24Hours Then
        deg = 15
        numHours = 24
    Else
        deg = 30
        numHours = 12
    End If

    Return 360.0F * currentTime.Hour / _
        numHours + deg * currentTime.Minute / 60.0F
End Function 'GetHourRotation

Private Function GetMinuteRotation() As Single
    Return 360.0F * currentTime.Minute / 60.0F
End Function 'GetMinuteRotation

Private Function GetSecondRotation() As Single
    Return 360.0F * currentTime.Second / 60.0F
End Function 'GetSecondRotation

Private Shared Function GetSin(ByVal degAngle As Single) As Single
    Return CSng(Math.Sin((Math.PI * degAngle / 180.0F)))
End Function 'GetSin

Private Shared Function GetCos(ByVal degAngle As Single) As Single
    Return CSng(Math.Cos((Math.PI * degAngle / 180.0F)))
End Function 'GetCos

Private Sub btnClockFormat_Click( _
    ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles btnClockFormat.Click

    If b24Hours Then
        btnClockFormat.Text = "24 Hours"
        b24Hours = False
    Else
```

Example 10-10. Clock face 2 in VB.NET (continued)

```
VB         btnClockFormat.Text = "12 Hours"
           b24Hours = True
           End If

           Me.Invalidate()

       End Sub
   End Class 'Form1
End Namespace
```

Creating the timer

One of the most significant changes in this version of the program is the use of a timer to tick off the seconds. You instantiate the timer in the constructor:

```
C#         System.Timers.Timer timer = new System.Timers.Timer();
```

```
VB         Dim timer As New System.Timers.Timer()
```

Set its event handler by passing in the name of the method you want called when the interval you'll specify has elapsed:

```
C#         timer.Elapsed += new System.Timers.ElapsedEventHandler(OnTimer);
```

```
VB         AddHandler timer.Elapsed, AddressOf OnTimer
```

The interval is set in milliseconds; in this case you'll update the timer every 500 milliseconds (every half second):

```
         timer.Interval = 500;
```

Finally, kick off the timer by enabling it:

```
         timer.Enabled = true;
```

Implementing OnTimer. The event handler you've passed to the timer's Elapsed event is OnTimer(). The implementation of OnTimer is similar to that of OnPaint: set the scale, draw the face, and then draw the hands. The latter operation occurs in a new method named DrawTime, discussed next:

```
C#         public void OnTimer(Object source, ElapsedEventArgs e
           {
               Graphics g = this.CreateGraphics();

               SetScale(g);
               DrawFace(g);
               DrawTime(g, false);

               g.Dispose();
           }
```

VB

```
Public Sub OnTimer(  
    ByVal source As Object, ByVal e As ElapsedEventArgs)  
    Dim g As Graphics = Me.CreateGraphics()  
  
    SetScale(g)  
    DrawFace(g)  
    DrawTime(g, False)  
  
    g.Dispose()  
End Sub 'OnTimer
```

The key difference between `OnTimer` and `OnPaint` is that the `EventArgs` structure passed to `OnTimer` does not have a `Graphics` object. You'll get one from the form by calling `CreateGraphics` (highlighted in the code snippet).

This `Graphics` object then invokes the same methods invoked in `OnPaint`. When you are done with the `Graphics` object obtained by `CreateGraphics`, you must dispose of it through a call to its `Dispose` method (also highlighted in the snippet).

DrawTime method

After `OnTimer` calls `DrawFace`, it calls `DrawTime` (`OnPaint` has been modified to call `DrawTime` as well). `DrawTime` is responsible for drawing the hands on the clock to correspond to the current time.

In the `DrawTime` method, you will first delete the hands from their current positions and then draw them in their new positions. You will draw the hands as lines and put an arrow at the end of the line to simulate an old fashioned clock's hand. Deleting the hands is accomplished by drawing the hands with a brush set to the color of the background (thus making them invisible).

Drawing the hands

You will draw the hands of the clock with a `Pen` object. The `Pen` class has properties and methods, described previously in Table 10-9.

Pass the pen to a drawing method, and that method determines how long a line to draw and what direction to draw in. The line you draw will have the `Color`, `Width`, and other characteristics you set with the `Pen`'s properties.

The `EndCap` property is of type `LineCap`, an enumeration listed in Table 10-12. In addition to the `ArrowAnchor` used in these examples, you can chose to create a `Round`, `Square`, `Triangle`, or `Flat` line cap, or you can create a `RoundAnchor`, `SquareAnchor`, or `NoAnchor`.

You instantiate a `Pen` with a color as follows:

C#

```
Pen myPen = new Pen(Color.Red);
```

VB

```
dim myPen as new Pen(Color.Red)
```

Deleting the existing line. Now that you have the necessary tools in hand, it is time to update the clock face. First, delete the hands from their old position. Start by creating three pens, one each to draw the hour, minute, and second hands. Each pen will use the background color:

```
C# Pen hourPen = new Pen(BackColor);  
    Pen minutePen = new Pen(BackColor);  
    Pen secondPen = new Pen(BackColor);
```

```
VB Dim hourPen As New Pen(BackColor)  
    Dim minutePen As New Pen(BackColor)  
    Dim secondPen As New Pen(BackColor)
```

Next, set the hour and minute pen to use an ArrowAnchor:

```
C# hourPen.EndCap = LineCap.ArrowAnchor;  
    minutePen.EndCap = LineCap.ArrowAnchor;
```

and set the width of the hour and minute pens:

```
C# hourPen.Width = 30;  
    minutePen.Width = 20;
```

You do not need to set the EndCap or Width of the second hand because you'll just draw a dot for the second hand (shown below). What you do need for drawing the second hand, however, is a brush:

```
C# Brush secondBrush = new SolidBrush(BackColor);
```

```
VB Dim secondBrush = New SolidBrush(BackColor)
```

Begin by deleting the second hand. To do so, you must determine the position in which to draw the second hand. Here you'll use an interesting approach. Rather than computing the x,y location of the second hand, assume that the second hand is always at 12 o'clock. How can this work? The answer is to rotate the world around the center of the clock face.

Picture a simple clock face with an x,y grid superimposed on it, as shown in Figure 10-8.

One way to draw a second hand at 2 o'clock is to compute the x,y coordinates of 2 o'clock (as you did when drawing the clock face). An alternative approach is to rotate the clock the appropriate number of degrees, and then draw the second hand straight up.

One way to think about this is to picture the clock face and a ruler, as shown in Figure 10-9. You can move the ruler to the right angle, or you can keep the ruler straight up and down and rotate the clock face under it. In the next example, use this second technique to draw the hands of the clock.

Create a method GetSecondRotation() to return a floating-point number, indicating how much the "paper" should be turned.

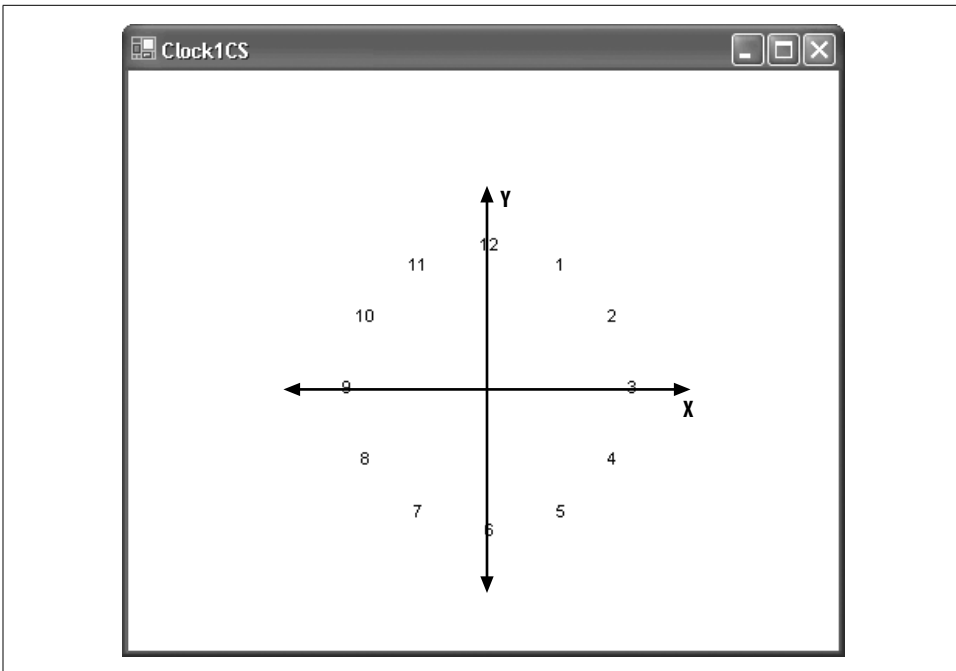


Figure 10-8. Drawing the clock face

```
C# float rotation = GetSecondRotation();
```

```
VB Dim rotation As Single
rotation = GetSecondRotation()
```

The helper method `GetSecondRotation` uses the current time member field. Notice that the `currentTime` field has not yet been updated, so it has the same “current time” that you had when you drew the hands.

Divide the current second by 60 (60 seconds per minute), and then multiply by 360 (360 degrees in a circle). For example, at 15 seconds past the minute, `GetSecondRotation()` will return 90 because $360 * 15 / 60 = 90$.

```
C# private float GetSecondRotation()
{
    return(360f * currentTime.Second / 60f);
}
```

```
VB Private Function GetSecondRotation() As Single
Return 360.0F * currentTime.Second / 60.0F
End Function 'GetSecondRotation
```

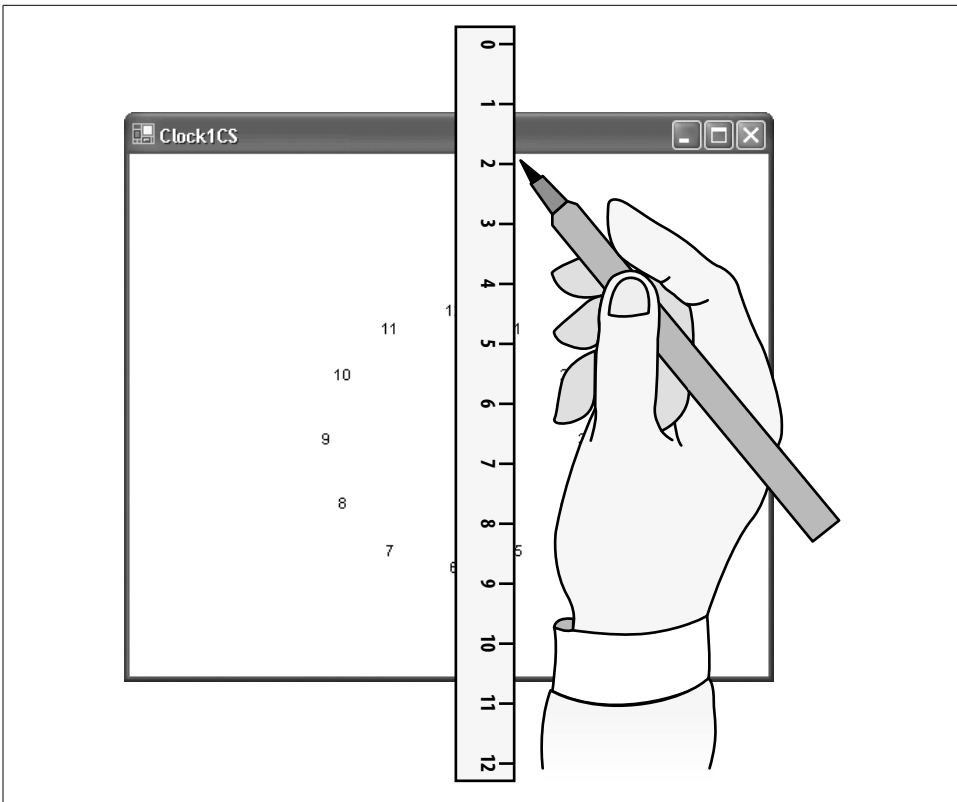


Figure 10-9. Paper and ruler

RotateTransform

You now know how much you want to rotate the world (i.e., rotate the paper under the ruler) to draw the second hand. The steps are:

1. Save the current state of the Graphics object
2. Rotate the world
3. Draw the second hand
4. Restore the state of the Graphics object

It is as if you spin your paper, draw the dot, and then spit it back to the way it was. The code snippet you need to accomplish this is (the VB.NET is virtually identical):

```
C# state = g.Save();  
g.RotateTransform(rotation);  
//...do stuff here  
g.Restore(state);
```

The transform method for rotating the world is called `RotateTransform`, and it takes a single argument: the number of degrees to rotate.

FillEllipse

The method you'll use to draw the dot representing the second hand is *FillEllipse*. This method of the Graphics object is overloaded; the version used here takes five parameters:

- The brush that will determine the color and texture of the ellipse
- The x coordinate of the upper-left-hand corner of the bounding rectangle
- The y coordinate of the upper-left-hand corner of the bounding rectangle
- The width of the bounding rectangle
- The height of the bounding rectangle

You'll use the brush you created earlier, named *secondBrush*. When you are deleting, *secondBrush* will be set to the background color. When you are drawing the second hand, it will be set to green.

The x and y coordinates of the second hand are determined so that the second hand is straight up from the origin, centered on the y axis (remember, you've turned the paper under the ruler. Now you should draw along the ruler).

The y coordinate is easy; you'll use the constant you've defined for the length of the second hand. Remember, however, that in this world, the y coordinates are negative above the origin, and since you want to draw straight up to 12 o'clock, you must use a negative value.

The x coordinate is just a bit trickier. The premise was that you'd just draw straight up, along the y axis. Unfortunately, this will place the upper-left-hand corner of the bounding rectangle along the y axis, and you'll want to center the ellipse on the y axis. You thus pass an x coordinate that is half the size of the bounding rectangle (e.g., 25) and set that negative so that the ball will be centered on the y axis.

Since you want your ellipse to be circular, the bounding rectangle will be square, with each side set to 50:

```
C#
const int EllipseSize = 50;
state = g.Save();
rotation = GetSecondRotation();
g.RotateTransform(rotation);
g.FillEllipse(
    secondBrush,
    -(EllipseSize/2),
    -secondLength,
    EllipseSize,
    EllipseSize);
g.Restore(state);
```

```
VB
Const EllipseSize As Single = 50
state = g.Save()
rotation = GetSecondRotation()
```

```

VB
g.RotateTransform(rotation)
g.FillEllipse( _
    secondBrush, _
    -(EllipseSize / 2), _
    -secondLength, _
    EllipseSize, _
    EllipseSize)
g.Restore(state)

```

Having drawn the second hand, go on to draw the minute and hour hand. If you redraw them both every second, however, the clock face flickers annoyingly. Therefore, redraw these two hands only if the minute has changed. To test this, compare the new time with the old time and determine whether the minute value has changed:

```

C#
DateTime newTime = DateTime.Now;
bool newMin = false; // has the minute changed?

if ( newTime.Minute != currentTime.Minute )
    newMin = true;

```

```

VB
Dim newTime As DateTime = DateTime.Now
Dim newMin As Boolean = False ' has the minute changed?

If newTime.Minute <> currentTime.Minute Then
    newMin = True
End If

```

You can then test the newMin Boolean value before updating the minute and hour hands:

```

C#
if ( newMin || forceDraw )
{
    // draw the minute and hour hands
}

```

```

VB
If newMin Or forceDraw Then
    ' draw the minute and hour hands
End If

```

The test is that *either* the minute has changed or the forceDraw parameter passed into the DrawTime method is true. This allows onPaint to ensure that the hands are drawn on a repaint by calling DrawTime and passing in true for the Boolean value.

The implementation of drawing the minute and hour hands is nearly identical to that for drawing the second hand. This time, however, rather than drawing an ellipse, you actually draw a line. You do so with the DrawLine method of the Graphics object, passing in a pen and four integer values.

The first two values represent the x,y coordinates of the origin of the line, and the second set of two values represent the x,y coordinates of the end of the line. In each

case, the origin of the line will be the center of the clock face, 0,0. The x coordinate of the end of the line will be 0 because you'll draw along the y axis. The y coordinate of the end of the line will be the length of the hour hand. Once again, because the y coordinates are negative above the origin, you'll pass it as a negative number.

The length of the hour and minute hands are defined at the top of the method, as is the distance from the origin for the ellipse representing the second hand:

```
float hourLength = FaceRadius * 0.5f;
float minuteLength = FaceRadius * 0.7f;
float secondLength = FaceRadius * 0.9f;
```



You may notice that you are drawing the line along the y axis (as you might run a pen along a ruler) rather than centered on the y axis. This keeps the code a bit simpler, but you are free to determine the width of the line and then to offset the drawing by that amount. This is left as an exercise for the obsessive-compulsive reader.

If the minute has advanced (or if forceDraw is true), you will determine the rotation for the minute, save the state of the Graphics object, rotate the world, draw the line, and restore the state of the Graphics object. You can then do the same thing for the hour hand:

```
C#
if ( newMin || forceDraw )
{
    rotation = GetMinuteRotation();
    state = g.Save();
    g.RotateTransform(rotation);
    g.DrawLine(minutePen,0,0,0,-minuteLength);
    g.Restore(state);

    rotation = GetHourRotation();
    state = g.Save();
    g.RotateTransform(rotation);
    g.DrawLine(hourPen,0,0,0,-hourLength);
    g.Restore(state);
}
```

```
VB
If newMin Or forceDraw Then
    rotation = GetMinuteRotation()
    state = g.Save()
    g.RotateTransform(rotation)
    g.DrawLine(minutePen, 0, 0, 0, -minuteLength)
    g.Restore(state)

    rotation = GetHourRotation()
    state = g.Save()
    g.RotateTransform(rotation)
    g.DrawLine(hourPen, 0, 0, 0, -hourLength)
    g.Restore(state)
End If
```

The two helper methods, `GetMinuteRotation` and `GetHourRotation`, simply determine the degrees to rotate the world for the current minute and hour. `GetMinuteRotation` is simple, it multiplies the 360 degrees of the clock by the current minute and divides by 60 (60 minutes in an hour):

```
C# private float GetMinuteRotation()
{
    return( 360f * currentTime.Minute / 60f );
}
```

```
VB Private Function GetMinuteRotation() As Single
    Return 360.0F * currentTime.Minute / 60.0F
End Function
```

The `GetHourRotation` method is more complicated because in this version you may have set the face to 24 hour mode, and the angle for the hour hand will be different if there are 24 hours around the clock face rather than 12.

Each hour will be 30 degrees from the previous hour if the clock face has 12 hours, or 15 degrees if the clock face has 24. To get the angle for the hour, multiply 360 by the current hour and divide by the number of hours (12 or 24) on the clock face.

You should also move the hour hand a bit more to allow for the number of minutes past the hour. For example, at 12:30 the hour hand should be halfway between the 12 and the 1.

To accomplish this adjustment, add another rotation computed by multiplying the number of degrees between hours (15 or 30) by the current number of minutes past the hour and dividing by 60:

```
C# private float GetHourRotation()
{
    float deg = b24Hours ? 15 : 30;
    float numHours = b24Hours ? 24 : 12;
    return( 360f * currentTime.Hour / numHours +
        deg * currentTime.Minute / 60f);
}
```

```
VB Private Function GetHourRotation() As Single
    Dim deg As Single
    Dim numHours As Single
    If b24Hours Then
        deg = 15
        numHours = 24
    Else
        deg = 30
        numHours = 12
    End If

    Return 360.0F * currentTime.Hour / _
        numHours + deg * currentTime.Minute / 60.0F
End Function 'GetHourRotation
```

Drawing the new time

Once you've done all the work shown so far, you've drawn the second hand, the minute hand, and the hour hand in the background color, effectively erasing them. Next, set the `currentTime` variable to the new time, and set the pen and brush colors to the colors you want to draw:

```
VB    currentTime = newTime

        hourPen.Color = Color.Red
        minutePen.Color = Color.Blue
        secondPen.Color = Color.Green
        secondBrush = New SolidBrush(Color.Green)
```

You are now ready to redraw these hands using the same technique shown above: save the state, rotate, draw the hand, and restore the state.



Notice the use of the Boolean variable `newMin`. Here's why it is required.

Imagine that you test the time when you are ready to erase the hands, but it is not a new minute. You thus do not erase the minute and hour hands, but test the time again when it is time to draw the hands with their correct colors. You might have just passed the minute mark, and now the minute values for current time and new time would be different, and you would draw the new hands without having erased them first. Suddenly the minute and hour hands get fatter.

You can avoid this bug by setting the `newMin` Boolean variable before erasing, and then using that Boolean when redrawing.

Implementing the 24 hour clock button

The event handler for the 24 hour clock button is straightforward: it toggles the `b24Hour` Boolean member variable and toggles the text. Finally, it invalidates the form so the clock is redrawn:

```
C#    private void btnClockFormat_Click(object sender, System.EventArgs e)
    {
        btnClockFormat.Text = b24Hours ? "24 Hour" : "12 Hour";
        b24Hours = ! b24Hours;
        this.Invalidate();
    }
```

```
VB    Private Sub btnClockFormat_Click( _
        ByVal sender As System.Object, _
        ByVal e As System.EventArgs) _
        Handles btnClockFormat.Click

        If b24Hours Then
            btnClockFormat.Text = "24 Hours"
            b24Hours = False
```

VB

```

Else
    btnClockFormat.Text = "12 Hours"
    b24Hours = True
End If

Me.Invalidate()

End Sub

```

The only remaining change you need to make to the code is to update the DrawFace method to draw either the 24 hour or the 12 hour clock face:

C#

```

private void DrawFace(Graphics g)
{
    Brush brush = new SolidBrush(ForeColor);
    Font font = new Font("Arial", 40);
    float x, y;

    int numHours = b24Hours ? 24 : 12;
    int deg = 360 / numHours;

    for (int i = 1; i <= numHours; i++)
    {
        x = GetCos(i*deg + 90) * FaceRadius;
        y = GetSin(i*deg + 90) * FaceRadius;

        StringFormat format = new StringFormat();
        format.Alignment = StringAlignment.Center;
        format.LineAlignment = StringAlignment.Center;

        g.DrawString(
            i.ToString(), font, brush, -x, -y,format);
    }
}

```

VB

```

Private Sub DrawFace(ByVal g As Graphics)
    Dim brush = New SolidBrush(ForeColor)
    Dim font As New Font("Arial", 40)
    Dim x, y As Single

    Dim numHours As Integer
    If b24Hours Then
        numHours = 24
    Else
        numHours = 12
    End If
    Dim deg As Integer = 360 / numHours
    Const FaceRadius As Integer = 450

    ' for each of the hours on the clock face
    Dim i As Integer
    For i = 1 To numHours

```

VB

```

        x = GetCos((i * deg + 90)) * FaceRadius
        y = GetSin((i * deg + 90)) * FaceRadius

        Dim format As New StringFormat()
        format.Alignment = StringAlignment.Center
        format.LineAlignment = StringAlignment.Center

        g.DrawString(i.ToString(), font, brush, -x, -y, format)
    Next i
End Sub 'DrawFace

```

The new code is shown in bold. The trick is to set the numHours variable to 12 or 24, based on the value of the member variable b24Hours. You then set the deg variable based on dividing the 360 degrees in the circle by the number of hours you are showing on the clock face. Then compute the Sin and Cosine value accordingly.

Drawing the Animated Date

In the third and final version of the program, you will add code to draw the date around the clock face and animate it. While you're at it, you'll also let the user click on the form to create a new center: by moving the clock's center to the location of the mouse when the user left-clicks. The complete source is shown in Examples 10-11 and 10-12.

Example 10-11. Final version clock face (CS)

C#

```

using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Timers;
using System.Windows.Forms;

namespace Clock3CS
{
    // Rename the class
    public class ClockFace : System.Windows.Forms.Form
    {
        // Required designer variable.
        private System.ComponentModel.IContainer components = null;

        private int FaceRadius = 450; // size of the clock face
        private bool b24Hours = false; // 24 hour clock face?
        private System.Windows.Forms.Button btnClockFormat;
        private DateTime currentTime; // used in more than one method

        // new
        private int xCenter; // center of the clock
        private int yCenter;
        private static int DateRadius = 600; // outer circumference for date
    }
}

```

Example 10-11. Final version clock face (CS) (continued)

C#

```
private static int Offset = 0; // for moving the text
Font font = new Font("Arial", 40); // use the same font throughout
private StringDraw sdToday; // the text to animate

public ClockFace()
{
    // Required for Windows Form Designer support
    InitializeComponent();

    // use the user's choice of colors
    BackColor = SystemColors.Window;
    ForeColor = SystemColors.WindowText;

    // *** begin new
    string today = System.DateTime.Now.ToLongDateString();
    today = " " + today.Replace(",","");

    // create a new stringdraw object with today's date
    sdToday = new StringDraw(today,this);
    currentTime = DateTime.Now;

    // set the current center based on the
    // client area
    xCenter = Width / 2;
    yCenter = Height / 2;

    // *** end new

    // update the clock by timer
    System.Timers.Timer timer = new System.Timers.Timer();
    timer.Elapsed += new System.Timers.ElapsedEventHandler(OnTimer);
    timer.Interval = 5; // shorter interval - more movement
    timer.Enabled = true;
}

protected override void OnPaint ( PaintEventArgs e )
{
    base.OnPaint(e);
    Graphics g = e.Graphics;
    SetScale(g);
    DrawFace(g);
    DrawTime(g,true); // force an update
}

// every time the timer event fires, update the clock
public void OnTimer(Object source, ElapsedEventArgs e)
```

Example 10-11. Final version clock face (CS) (continued)

C#

```
{
    Graphics g = this.CreateGraphics();

    SetScale(g);
    DrawFace(g);
    DrawTime(g,false);
    DrawDate(g);
    g.Dispose();
}

#region Windows Form Designer generated code
#endregion

[STAThread]
static void Main()
{
    Application.Run(new ClockFace());
}

private void SetScale(Graphics g)
{
    // if the form is too small, do nothing
    if ( Width == 0 || Height == 0 )
        return;

    // set the origin at the center
    g.TranslateTransform(xCenter, yCenter); // use the members vars

    // set inches to the minimum of the width
    // or height divided by the dots per inch
    float inches = Math.Min(Width / g.DpiX, Height / g.DpiY);

    // set the scale to a grid of 2000 by 2000 units
    g.ScaleTransform(
        inches * g.DpiX / 2000, inches * g.DpiY / 2000);
}

private void DrawFace(Graphics g)
{
    // numbers are in forecolor except flash number in green
    // as the seconds go by.
    Brush brush = new SolidBrush(ForeColor);
    float x, y;

    // new code
    int numHours = b24Hours ? 24 : 12;
    int deg = 360 / numHours;

    // for each of the hours on the clock face
    for (int i = 1; i <= numHours; i++)
```

Example 10-11. Final version clock face (CS) (continued)

C#

```
{
    // i = hour 30 degrees = offset per hour
    // +90 to make 12 straight up
    x = GetCos(i*deg + 90) * FaceRadius;
    y = GetSin(i*deg + 90) * FaceRadius;

    StringFormat format = new StringFormat();
    format.Alignment = StringAlignment.Center;
    format.LineAlignment = StringAlignment.Center;

    g.DrawString(
        i.ToString(), font, brush, -x, -y,format);
} // end for loop
} // end drawFace

private void DrawTime(Graphics g, bool forceDraw)
{
    // length of the hands
    float hourLength = FaceRadius * 0.5f;
    float minuteLength = FaceRadius * 0.7f;
    float secondLength = FaceRadius * 0.9f;

    // set to back color to erase old hands first
    Pen hourPen = new Pen(BackColor);
    Pen minutePen = new Pen(BackColor);
    Pen secondPen = new Pen(BackColor);

    // set the arrow heads
    hourPen.EndCap = LineCap.ArrowAnchor;
    minutePen.EndCap = LineCap.ArrowAnchor;

    // hour hand is thicker
    hourPen.Width = 30;
    minutePen.Width = 20;

    // second hand
    Brush secondBrush = new SolidBrush(BackColor);
    const int EllipseSize = 50;

    GraphicsState state; // to to protect and to serve

    // 1 - delete the old time

    // delete the old second hand
    // figure out how far around to rotate to draw the second hand
    // save the current state, rotate, draw and then restore the state
    float rotation = GetSecondRotation();
    state = g.Save();
```

Example 10-11. Final version clock face (CS) (continued)

C#

```
g.RotateTransform(rotation);
g.FillEllipse(
    secondBrush,
    -(EllipseSize/2),
    -secondLength,
    EllipseSize,
    EllipseSize);
g.Restore(state);

DateTime newTime = DateTime.Now;
bool newMin = false; // has the minute changed?

// if the minute has changed, set the flag
if ( newTime.Minute != currentTime.Minute )
    newMin = true;

// if the minute has changed or you must draw anyway then you
// must first delete the old minute and hour hand
if ( newMin || forceDraw )
{
    // figure out how far around to rotate to draw the minute hand
    // save the current state, rotate, draw and
    // then restore the state
    rotation = GetMinuteRotation();
    state = g.Save();
    g.RotateTransform(rotation);
    g.DrawLine(minutePen,0,0,0,-minuteLength);
    g.Restore(state);

    // figure out how far around to rotate to draw the hour hand
    // save the current state, rotate, draw and
    // then restore the state
    rotation = GetHourRotation();
    state = g.Save();
    g.RotateTransform(rotation);
    g.DrawLine(hourPen,0,0,0,-hourLength);
    g.Restore(state);
}

// step 2 - draw the new time
currentTime = newTime;

hourPen.Color = Color.Red;
minutePen.Color = Color.Blue;
secondPen.Color = Color.Green;
secondBrush = new SolidBrush(Color.Green);

// draw the new second hand
// figure out how far around to rotate to draw the second hand
// save the current state, rotate, draw and then restore the state
```

Example 10-11. Final version clock face (CS) (continued)

C#

```
state = g.Save();
rotation = GetSecondRotation();
g.RotateTransform(rotation);
g.FillEllipse(
    secondBrush,
    -(EllipseSize/2),
    -secondLength,
    EllipseSize,
    EllipseSize);
g.Restore(state);

// if the minute has changed or you must draw anyway then you
// must draw the new minute and hour hand
if ( newMin || forceDraw )
{
    // figure out how far around to rotate to draw the minute hand
    // save the current state, rotate, draw and
    // then restore the state
    state = g.Save();
    rotation = GetMinuteRotation();
    g.RotateTransform(rotation);
    g.DrawLine(minutePen,0,0,0,-minuteLength);
    g.Restore(state);

    // figure out how far around to rotate to draw the hour hand
    // save the current state, rotate, draw and
    // then restore the state
    state = g.Save();
    rotation = GetHourRotation();
    g.RotateTransform(rotation);
    g.DrawLine(hourPen,0,0,0,-hourLength);
    g.Restore(state);
}
}

// determine the rotation to draw the hour hand
private float GetHourRotation()
{
    // degrees depend on 24 vs. 12 hour clock
    float deg = b24Hours ? 15 : 30;
    float numHours = b24Hours ? 24 : 12;
    return( 360f * currentTime.Hour / numHours +
        deg * currentTime.Minute / 60f);
}

private float GetMinuteRotation()
{
    return( 360f * currentTime.Minute / 60f );
}

private float GetSecondRotation()
{

```

Example 10-11. Final version clock face (CS) (continued)

C#

```
        return(360f * currentTime.Second / 60f);
    }

    private static float GetSin(float degAngle)
    {
        return (float) Math.Sin(Math.PI * degAngle / 180f);
    }

    private static float GetCos(float degAngle)
    {
        return (float) Math.Cos(Math.PI * degAngle / 180f);
    }

    private void btnClockFormat_Click(object sender, System.EventArgs e)
    {
        btnClockFormat.Text = b24Hours ? "24 Hour" : "12 Hour";
        b24Hours = ! b24Hours;
        this.Invalidate();
    }

    private void DrawDate(Graphics g)
    {
        Brush brush = new SolidBrush(ForeColor);
        sdToday.DrawString(g,brush);
    }

    private void ClockFace_MouseDown(
        object sender, System.Windows.Forms.MouseEventArgs e)
    {
        xCenter = e.X;
        yCenter = e.Y;
        this.Invalidate();
    }

    // each letter in the outer string knows how to draw itself
    private class LtrDraw
    {
        char myChar;        // the actual letter i draw
        float x;           // current x coordinate
        float y;           // current y coordinate
        float oldx;        // old x coordinate (to delete)
        float oldy;        // old y coordinate (to delete)

        // constructor
        public LtrDraw(char c)
        {
            myChar = c;
        }
    }
}
```

Example 10-11. Final version clock face (CS) (continued)

C#

```
// property for X coordinate
public float X
{
    get { return x; }
    set { oldx = x; x = value; }
}

// property for Y coordinate
public float Y
{
    get { return y; }
    set { oldy = y; y = value; }
}

// get total width of the string
public float GetWidth(Graphics g, Font font)
{
    SizeF stringSize = g.MeasureString(myChar.ToString(),font);
    return stringSize.Width;
}

// get total height of the string
public float GetHeight(Graphics g, Font font)
{
    SizeF stringSize = g.MeasureString(myChar.ToString(),font);
    return stringSize.Height;
}

// get the font from the control and draw the current character
// First delete the old and then draw the new
public void DrawString(Graphics g, Brush brush, ClockFace cf)
{
    Font font = cf.font;
    Brush blankBrush = new SolidBrush(cf.BackColor);
    g.DrawString(myChar.ToString(),font,blankBrush,oldx,oldy);
    g.DrawString(myChar.ToString(),font,brush,x,y);
}

}

// holds an array of LtrDraw objects
// and knows how to tell them to draw
private class StringDraw
{
    ArrayList theString = new ArrayList();
    LtrDraw l;
    ClockFace theControl;

    // constructor takes a string, populates the array
    // and stashes away the calling control (ClockFace)
    public StringDraw(string s, ClockFace theControl)
```

Example 10-11. Final version clock face (CS) (continued)

C#

```
{
    this.theControl = theControl;
    foreach (char c in s)
    {
        l = new LtrDraw(c);
        theString.Add(l);
    }
}

// divide the circle by the number of letters
// and draw each letter in position
public void DrawString(Graphics g, Brush brush)
{
    int angle = 360 / theString.Count;
    int counter = 0;

    foreach (LtrDraw theLtr in theString)
    {
        // 1. To find the X coordinate, take the Cosine of the angle
        // and multiply by the radius.
        // 2. To compute the angle, start with the base angle
        // (360 divided by the number of letters)
        // and multiply by letter position.
        // Thus if each letter is 10 degrees, and this is the third
        // letter, you get 30 degrees.
        // Add 90 to start at 12 O'clock.
        // Each time through, subtract the clockFace offset to move
        // the entire string around the clock on each timer call
        float newX = GetCos(
            angle * counter + 90 -
            ClockFace.Offset) * ClockFace.DateRadius ;
        float newY = GetSin(
            angle * counter + 90 -
            ClockFace.Offset) * ClockFace.DateRadius ;
        theLtr.X =
            newX - (theLtr.GetWidth(g,theControl.font) / 2);
        theLtr.Y =
            newY - (theLtr.GetHeight(g,theControl.font) / 2);
        counter++;
        theLtr.DrawString(g,brush,theControl);
    }
    ClockFace.Offset += 1; // rotate the entire string
}
}
} // end class
} // end namespace
```

Example 10-12. Final version clock face (VB.NET)

VB

```
Imports System
Imports System.Collections
Imports System.ComponentModel
Imports System.Data
```

Example 10-12. Final version clock face (VB.NET) (continued)

```
Imports System.Drawing
Imports System.Drawing.Drawing2D
Imports System.Timers
Imports System.Windows.Forms

Namespace Clock3VB

    Public Class ClockFace
        Inherits System.Windows.Forms.Form

        Private FaceRadius As Integer = 450 ' size of the clock face
        Private b24Hours As Boolean = False ' 24 hour clock face?
        Private currentTime As DateTime ' used in more than one method
        ' new
        Private xCenter As Integer ' center of the clock
        Private yCenter As Integer
        ' outer circumference for date
        Private Shared DateRadius As Integer = 600
        Private Shared offset As Integer = 0 ' for moving the text
        ' use the same font throughout
        Private myFont As New font("Arial", 40)
        Private sdToday As StringDraw

        Public Sub New()
            ' Required for Windows Form Designer support
            InitializeComponent()

            ' use the user's choice of colors
            BackColor = SystemColors.Window
            ForeColor = SystemColors.WindowText

            ' *** begin new code
            Dim today As String = System.DateTime.Now.ToLongDateString()
            today = " " + today.Replace(",", "")

            ' create a new stringdraw object with today's date
            sdToday = New StringDraw(today, Me)
            currentTime = DateTime.Now

            ' set the current center based on the
            ' client area
            xCenter = Width / 2
            yCenter = Height / 2

            ' *** end new code

            ' update the clock by timer
            Dim timer As New System.Timers.Timer()
            AddHandler timer.Elapsed, AddressOf OnTimer
        End Sub
    End Class
End Namespace
```

Example 10-12. Final version clock face (VB.NET) (continued)

VB

```
timer.Interval = 5 ' shorter interval - more movement
timer.Enabled = True
End Sub 'New
```

```
' every time the timer event fires, update the clock
Public Sub OnTimer( _
    ByVal source As Object, ByVal e As ElapsedEventArgs)
    Dim g As Graphics = Me.CreateGraphics()

    SetScale(g)
    DrawFace(g)
    DrawTime(g, False)
    DrawDate(g)
    g.Dispose()
End Sub 'OnTimer
```

```
#Region " Windows Form Designer generated code "
```

```
#End Region
```

```
Protected Overrides Sub OnPaint(ByVal e As PaintEventArgs)
    myBase.OnPaint(e)
    Dim g As Graphics = e.Graphics
    SetScale(g)
    DrawFace(g)
    DrawTime(g, True) ' force an update
End Sub 'OnPaint
```

```
Private Sub SetScale(ByVal g As Graphics)
    ' if the form is too small, do nothing
    If Width = 0 Or Height = 0 Then
        Return
    End If
    ' set the origin at the center
    g.TranslateTransform(xCenter, yCenter) ' use the members vars
    ' set inches to the minimum of the width
    ' or height divided by the dots per inch
    Dim inches As Single = _
        Math.Min(Width / g.DpiX, Height / g.DpiY)

    ' set the scale to a grid of 2000 by 2000 units
    g.ScaleTransform( _
        inches * g.DpiX / 2000, inches * g.DpiY / 2000)
End Sub 'SetScale
```

```
Private Sub DrawFace(ByVal g As Graphics)
    ' numbers are in forecolor except flash number in green
    ' as the seconds go by.
```

Example 10-12. Final version clock face (VB.NET) (continued)

VB

```
Dim brush = New SolidBrush(ForeColor)
Dim x, y As Single

' new code

Dim numHours As Integer
If (b24Hours) Then
    numHours = 24
Else
    numHours = 12
End If

Dim deg As Integer = 360 / numHours

' for each of the hours on the clock face
Dim i As Integer
For i = 1 To numHours
    ' i = hour 30 degrees = offset per hour
    ' +90 to make 12 straight up
    x = GetCos((i * deg + 90)) * FaceRadius
    y = GetSin((i * deg + 90)) * FaceRadius

    Dim format As New StringFormat()
    format.Alignment = StringAlignment.Center
    format.LineAlignment = StringAlignment.Center

    g.DrawString(i.ToString(), myFont, brush, -x, -y, format)
Next i
End Sub 'DrawFace

' end for loop
' end drawFace

Private Sub DrawTime( _
    ByVal g As Graphics, ByVal forceDraw As Boolean)

    ' length of the hands
    Dim hourLength As Single = FaceRadius * 0.5F
    Dim minuteLength As Single = FaceRadius * 0.7F
    Dim secondLength As Single = FaceRadius * 0.9F

    ' set to back color to erase old hands first
    Dim hourPen As New Pen(BackColor)
    Dim minutePen As New Pen(BackColor)
    Dim secondPen As New Pen(BackColor)

    ' set the arrow heads
    hourPen.EndCap = LineCap.ArrowAnchor
    minutePen.EndCap = LineCap.ArrowAnchor

    ' hour hand is thicker
    hourPen.Width = 30
    minutePen.Width = 20
```

Example 10-12. Final version clock face (VB.NET) (continued)

VB

```
' second hand
Dim secondBrush = New SolidBrush(BackColor)
Const EllipseSize As Integer = 50
Dim halfEllipseSize As Integer = EllipseSize / 2

Dim state As GraphicsState ' to to protect and to serve

' 1 - delete the old time
' delete the old second hand
' figure out how far around to rotate to draw the second hand
' save the current state, rotate, draw
' and then restore the state
Dim rotation As Single = GetSecondRotation()
state = g.Save()
g.RotateTransform(rotation)

g.FillEllipse( _
    secondBrush, -(halfEllipseSize), _
    -secondLength, EllipseSize, EllipseSize)
g.Restore(state)

Dim newTime As DateTime = DateTime.Now
Dim newMin As Boolean = False ' has the minute changed?
' if the minute has changed, set the flag
If newTime.Minute <> currentTime.Minute Then
    newMin = True
End If

' if the minute has changed or you must draw anyway then you
' must first delete the old minute and hour hand
If newMin Or forceDraw Then

    ' figure out how far around to rotate to
    ' draw the minute hand
    ' save the current state, rotate, draw
    ' and then restore the state
    rotation = GetMinuteRotation()
    state = g.Save()
    g.RotateTransform(rotation)
    g.DrawLine(minutePen, 0, 0, 0, -minuteLength)
    g.Restore(state)

    ' figure out how far around to rotate to draw the hour hand
    ' save the current state, rotate, draw
    ' and then restore the state
    rotation = GetHourRotation()
    state = g.Save()
    g.RotateTransform(rotation)
    g.DrawLine(hourPen, 0, 0, 0, -hourLength)
    g.Restore(state)
End If
```

Example 10-12. Final version clock face (VB.NET) (continued)

VB

```
' step 2 - draw the new time
currentTime = newTime

hourPen.Color = Color.Red
minutePen.Color = Color.Blue
secondPen.Color = Color.Green
secondBrush = New SolidBrush(Color.Green)

' draw the new second hand
' figure out how far around to rotate to draw the second hand
' save the current state, rotate, draw
' and then restore the state
state = g.Save()
rotation = GetSecondRotation()
g.RotateTransform(rotation)
g.FillEllipse( _
    secondBrush, -(halfEllipseSize), _
    -secondLength, EllipseSize, EllipseSize)
g.Restore(state)

' if the minute has changed or you must draw anyway then you
' must draw the new minute and hour hand
If newMin Or forceDraw Then

    ' figure out how far around to rotate to
    ' draw the minute hand
    ' save the current state, rotate, draw
    ' and then restore the state
    state = g.Save()
    rotation = GetMinuteRotation()
    g.RotateTransform(rotation)
    g.DrawLine(minutePen, 0, 0, 0, -minuteLength)
    g.Restore(state)

    ' figure out how far around to rotate to draw the hour hand
    ' save the current state, rotate, draw
    ' and then restore the state
    state = g.Save()
    rotation = GetHourRotation()
    g.RotateTransform(rotation)
    g.DrawLine(hourPen, 0, 0, 0, -hourLength)
    g.Restore(state)
End If
End Sub 'DrawTime

' determine the rotation to draw the hour hand
Private Function GetHourRotation() As Single
    ' degrees depend on 24 vs. 12 hour clock
    Dim deg As Single
    Dim numHours As Single
    If (b24Hours) Then
```

Example 10-12. Final version clock face (VB.NET) (continued)

VB

```
        deg = 15
        numHours = 24
    Else
        deg = 30
        numHours = 12
    End If
    Return 360.0F * currentTime.Hour / _
        numHours + deg * currentTime.Minute / 60.0F
End Function 'GetHourRotation

Private Function GetMinuteRotation() As Single
    Return 360.0F * currentTime.Minute / 60.0F
End Function 'GetMinuteRotation

Private Function GetSecondRotation() As Single
    Return 360.0F * currentTime.Second / 60.0F
End Function 'GetSecondRotation

Private Shared Function GetSin(ByVal degAngle As Single) As Single
    Return CSng(Math.Sin((Math.PI * degAngle / 180.0F)))
End Function 'GetSin

Private Shared Function GetCos(ByVal degAngle As Single) As Single
    Return CSng(Math.Cos((Math.PI * degAngle / 180.0F)))
End Function 'GetCos

Private Sub btnClockFormat_Click( _
    ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles btnClockFormat.Click
    If (b24Hours) Then
        btnClockFormat.Text = "24 Hour"
    Else
        btnClockFormat.Text = "12 Hour"
    End If
    b24Hours = Not b24Hours
    Me.Invalidate()
End Sub 'btnClockFormat_Click

Private Sub DrawDate(ByVal g As Graphics)
    Dim brush = New SolidBrush(ForeColor)
    sdToday.DrawString(g, brush)
End Sub 'DrawDate
```

Example 10-12. Final version clock face (VB.NET) (continued)

VB

```
Private Sub ClockFace_MouseDown( _
    ByVal sender As Object, _
    ByVal e As System.Windows.Forms.MouseEventArgs) _
    Handles MyBase.MouseDown
    xCenter = e.X
    yCenter = e.Y
    Me.Invalidate()
End Sub 'ClockFace_MouseDown

-

' each letter in the outer string knows how to draw itself
Private Class LtrDraw
    Private myChar As Char ' the actual letter i draw
    Private _x As Single ' current x coordinate
    Private _y As Single ' current y coordinate
    Private oldx As Single ' old x coordinate (to delete)
    Private oldy As Single
        ' old y coordinate (to delete)

    ' constructor
    Public Sub New(ByVal c As Char)
        myChar = c
    End Sub 'New

    ' property for X coordinate

    Public Property X() As Single
        Get
            Return _x
        End Get
        Set(ByVal Value As Single)
            oldx = _x
            _x = Value
        End Set
    End Property

    ' property for Y coordinate

    Public Property Y() As Single
        Get
            Return _y
        End Get
        Set(ByVal Value As Single)
            oldy = _y
            _y = Value
        End Set
    End Property

    ' get total width of the string
    Public Function GetWidth( _
        ByVal g As Graphics, ByVal myFont As Font) As Single
        Dim stringSize As SizeF = _
```

Example 10-12. Final version clock face (VB.NET) (continued)

VB

```
        g.MeasureString(myChar.ToString(), myFont)
    Return stringSize.Width
End Function 'GetWidth

' get total height of the string
Public Function GetHeight( _
    ByVal g As Graphics, ByVal myFont As Font) As Single
    Dim stringSize As SizeF = _
        g.MeasureString(myChar.ToString(), myFont)
    Return stringSize.Height
End Function 'GetHeight

' get the font from the control and draw the current character
' First delete the old and then draw the new
Public Sub DrawString( _
    ByVal g As Graphics, ByVal brush As Brush, _
    ByVal ctrl As ClockFace)
    Dim myFont As Font = ctrl.myFont
    Dim blankBrush = New SolidBrush(ctrl.BackColor)
    g.DrawString( _
        myChar.ToString(), myFont, blankBrush, oldx, oldy)
    g.DrawString(myChar.ToString(), myFont, brush, X, Y)
End Sub 'DrawString
End Class 'LtrDraw

-

' holds an array of LtrDraw objects
' and knows how to tell them to draw
Private Class StringDraw
    Private theString As New ArrayList()
    Private l As LtrDraw
    Private theControl As ClockFace

    ' constructor takes a string, populates the array
    ' and stashes away the calling control (ClockFace)
    Public Sub New( _
        ByVal s As String, ByVal theControl As ClockFace)
        Me.theControl = theControl
        Dim c As Char
        For Each c In s
            l = New LtrDraw(c)
            theString.Add(l)
        Next c
    End Sub 'New

    ' divide the circle by the number of letters
    ' and draw each letter in position
```

Example 10-12. Final version clock face (VB.NET) (continued)

VB

```
Public Sub DrawString( _
    ByVal g As Graphics, ByVal brush As Brush)
    Dim angle As Integer = 360 / theString.Count
    Dim counter As Integer = 0

    Dim theLtr As LtrDraw
    For Each theLtr In theString
        ' 1. To find the X coordinate,
        ' take the Cosine of the angle
        ' and multiply by the radius.
        ' 2. To compute the angle, start with the base angle
        ' (360 divided by the number of letters)
        ' and multiply by letter position.
        ' Thus if each letter is 10 degrees,
        ' and this is the third
        ' letter, you get 30 degrees.
        ' Add 90 to start at 12 O'clock.
        ' Each time through, subtract the clockFace
        ' offset to move the entire string around
        ' the clock on each timer call
        Dim newX As Single = _
            GetCos((angle * counter + 90 - ClockFace.offset)) _
            * ClockFace.DateRadius
        Dim newY As Single = _
            GetSin((angle * counter + 90 - ClockFace.offset)) _
            * ClockFace.DateRadius
        theLtr.X = newX - _
            theLtr.GetWidth(g, theControl.myFont) / 2
        theLtr.Y = newY - _
            theLtr.GetHeight(g, theControl.myFont) / 2
        counter += 1
        theLtr.DrawString(g, brush, theControl)
    Next theLtr
    ClockFace.offset += 1 ' rotate the entire string
End Sub 'DrawString
End Class 'StringDraw
End Class 'ClockFace
End Namespace 'Clock3CS ' end class
```

Animating the string

In the previous examples, you saw two ways to manage drawing text at a specific location. In the first, you determined the x,y coordinates and then used the DrawString method to draw the characters at that location (clock face). In the second, you rotated the world a set rotation, and then used DrawString to draw each text character to a specific location (e.g., centered on the y axis, a fixed distance from the origin, as seen when using DrawTime).

In the next example, however, you want the date to move around the clock face, and more importantly, you want the letters to act as cars on a Ferris Wheel, maintaining their up-down orientation as they rotate around the center.

Ferris Wheel

The Ferris Wheel was invented by George W. Ferris, a bridge builder from Pittsburgh, Pennsylvania, and shown at the 1893 World's Columbian Exposition in Chicago. The original wheel was supported by twin steel towers, each standing 140 feet tall; its 45 foot axel was the largest piece of forged steel in the world. The wheel was 250 feet in diameter, and its circumference was 825 feet. It stood 264 feet in the air and was powered by two 1,000 horsepower engines. The wheel had 36 wooden cars, each capable of holding 60 people, keeping them upright at all times. Over 1.5 million people rode the original Ferris Wheel at the Chicago fair.

The LtrDraw class. To accomplish this design goal, each letter in the date will be encapsulated by an instance of the LtrDraw class that you will define. The LtrDraw class will be used only by methods of ClockFace, so LtrDraw will be declared as a nested class within the ClockFace class.

```
C# public class ClockFace : System.Windows.Forms.Form
    {
        //...
        private class LtrDraw
        {
            // ...
        } // end nested class
    } // end outer class
```

This class will have, as member variables, both the character you want to draw and the x,y coordinates of where to draw it. In fact, the LtrDraw instance will know two sets of x,y coordinates: where the letter was (so you can erase the old letter) and where it is (so you can draw the letter in its new location):

```
C# private class LtrDraw
    {
        char myChar;
        float x;
        float y;
        float oldx;
        float oldy;
```

```
VB Private Class LtrDraw
    Private myChar As Char
    Private _x As Single
    Private _y As Single
    Private oldx As Single
    Private oldy As Single
```

The LtrDraw constructor initializes the myChar member variable:

```
C#
public LtrDraw(char c)
{
    myChar = c;
}
```

```
VB
Public Sub New(ByVal c As Char)
    myChar = c
End Sub 'New
```

The x,y coordinates are accessed through properties. The get accessor just returns the member variable's value, but the set accessor first stores the current value in the oldx/oldy members:

```
C#
public float X
{
    get { return x; }
    set { oldx = x; x = value; }
}

public float Y
{
    get { return y; }
    set { oldy = y; y = value; }
}
```

```
VB
Public Property X() As Single
    Get
        Return _x
    End Get
    Set(ByVal Value As Single)
        oldx = _x
        _x = Value
    End Set
End Property

Public Property Y() As Single
    Get
        Return _y
    End Get
    Set(ByVal Value As Single)
        oldy = _y
        _y = Value
    End Set
End Property
```

The LtrDraw class also provides methods that return the letter's Width and Height. These two methods delegate the actual measurement to the MeasureString method of the Graphics object, passing in the character the object holds in the myChar member variable and the font that is passed in to the method:

```

C#
public float GetWidth(Graphics g, Font font)
{
    SizeF stringSize = g.MeasureString(myChar.ToString(),font);
    return stringSize.Width;
}
public float GetHeight(Graphics g, Font font)
{
    SizeF stringSize = g.MeasureString(myChar.ToString(),font);
    return stringSize.Height;
}

```

```

VB
Public Function GetWidth( _
    ByVal g As Graphics, ByVal myFont As Font) As Single
    Dim stringSize As SizeF = _
        g.MeasureString(myChar.ToString(), myFont)
    Return stringSize.Width
End Function 'GetWidth

' get total height of the string
Public Function GetHeight( _
    ByVal g As Graphics, ByVal myFont As Font) As Single
    Dim stringSize As SizeF = _
        g.MeasureString(myChar.ToString(), myFont)
    Return stringSize.Height
End Function 'GetHeight

```

Finally, the `LtrDraw` class knows how to draw the letter via the `DrawString` method, given a `Brush` and a reference to the `ClockFace` object:

```

C#
public void DrawString(Graphics g, Brush brush, ClockFace cf)
{

```

The first task is to get a reference to the font held by the `ClockFace` as a member variable:

```

C#
    Font font = cf.font;

```

Next, create a blank brush and use it to delete the character from its old position:

```

C#
    Brush blankBrush = new SolidBrush(cf.BackgroundColor);
    g.DrawString(myChar.ToString(), font, blankBrush, oldx, oldy);

```

Finally, you are ready to draw the character in the new position, using the font you've extracted from the `ClockFace` and the brush you were given:

```

C#
    g.DrawString(myChar.ToString(), font, brush, x, y);
}

```

```

VB
Public Sub DrawString( _
    ByVal g As Graphics, ByVal brush As Brush, ByVal ctrl As ClockFace)
    Dim myFont As Font = ctrl.myFont
    Dim blankBrush = New SolidBrush(ctrl.BackgroundColor)

```

```

VB | g.DrawString(myChar.ToString(), myFont, blankBrush, oldx, oldy)
    g.DrawString(myChar.ToString(), myFont, brush, X, Y)
    End Sub 'DrawString

```

The StringDraw class. The LtrDraw class encapsulates a single letter. For the entire string, create a collection class to hold an array of LtrDraw objects. The StringDraw class uses an ArrayList to allow you to build up an array of LtrDraw objects and it holds a reference to the ClockFace object. StringDraw will be a nested class within ClockFace as well:

```

C# | private class StringDraw
    {
        ArrayList theString = new ArrayList();
        LtrDraw l;
        ClockFace theControl;
    }

```

```

VB | Private Class StringDraw
    Private theString As New ArrayList()
    Private l As LtrDraw
    Private theControl As ClockFace

```

Use the member variable l, the reference to a LtrDraw object, in the constructor to create instances of LtrDraw that you can add to the collection:

```

C# | public StringDraw(string s, ClockFace theControl)
    {
        this.theControl = theControl;
        foreach (char c in s)
        {
            l = new LtrDraw(c);
            theString.Add(l);
        }
    }

```

```

VB | Public Sub New(ByVal s As String, ByVal theControl As ClockFace)
    Me.theControl = theControl
    Dim c As Char
    For Each c In s
        l = New LtrDraw(c)
        theString.Add(l)
    Next c
    End Sub 'New

```

You are passed a string and a reference to a ClockFace object. Stash the reference in the member variable theControl. Then treat the string as an array of characters, and iterate through the array using the foreach (for each) construct. For each letter you retrieve from the string, create an instance of the LtrDraw class, and then add that instance to the ArrayList member.



Reusing the `LtrDraw` reference (`l`) is safe because a reference to the new object is kept in the `ArrayList`.

The only method in the `StringDraw` class is cleverly named `DrawString`. This method takes two arguments: a `Graphics` object and a `Brush`.

```
C# public void DrawString(Graphics g, Brush brush)
{
```

```
VB Public Sub DrawString(ByVal g As Graphics, ByVal brush As Brush)
```

This method first sets the angle by which each letter will be separated. Ask the string for the count of characters and use that value to divide the 360 degrees of the circle into equal increments:

```
C# int angle = 360 / theString.Count;
```

```
VB Dim angle As Integer = 360 / theString.Count
```

Your job now is to iterate through the members of the `ArrayList`. For each `LtrDraw` object, compute the new `x` and `y` coordinates.

Do so by multiplying the angle value computed above by what amounts to the `i`-based index of the letter (that is, 1 for the second letter, 2 for the third, and so forth). Then add 90 to start the string at 12 o'clock (this is not strictly necessary, since the string will rotate around the clock face). Take the cosine of this value (using your old friend `GetCos`, which converts the angle to radians and then returns the cosine of that angle), and multiply by the constant `DateRadius` defined in the `ClockFace` class:

```
C# float newX =
    GetCos(angle * counter + 90) * ClockFace.DateRadius ;
```

To make the string move, however, you have one more task. In the `ClockFace` class, declare a static (shared) member variable named `offset`. Modify your computation of the angle to subtract this value from the computed angle:

```
C# float newX =
    GetCos(angle * counter + 90 - ClockFace.Offset) * ClockFace.DateRadius ;
```

Each time this method is invoked, you'll increment the `offset` value so that each time you run this method, the string will be drawn using an angle one degree less than the previous time.

You can compute the new `y` coordinate in much the same way:

```
C# float newY =
    GetSin(angle * counter + 90 - ClockFace.Offset) * ClockFace.DateRadius ;
```

```

VB Dim newX As Single = _
      GetCos((angle * counter + 90 - ClockFace.offset)) _
      * ClockFace.DateRadius
      Dim newY As Single = _
      GetSin((angle * counter + 90 - ClockFace.offset)) _
      * ClockFace.DateRadius

```

Once again, however, you've computed the upper-left-hand corner of the bounding rectangle for the character you are going to draw. To center the character at this location, you must compute the width and height of the character and adjust your coordinates accordingly:

```

C# theLtr.X =
      newX - (theLtr.GetWidth(g,theControl.font) / 2);
      theLtr.Y =
      newY - (theLtr.GetHeight(g,theControl.font) / 2);

```

That accomplished, increment the counter:

```

C# counter++;

```

```

VB counter += 1

```

and you are ready to tell the LtrDraw object to draw itself:

```

theLtr.DrawString(g,brush,theControl);

```

Once the loop is completed, increment the static Offset member of the ClockFace:

```

C# ClockFace.Offset += 1;

```

To encourage the date to move around the clock face quickly and smoothly, change the timer interval from 500 milliseconds to 50 milliseconds. Do this in the constructor, where you'll make a few other changes as well, shown below.

New member variables. Before examining the constructor, you'll need to add six new member variables.

The xCenter and yCenter variables will hold the x and y coordinates of the center of the clock.

```

C# private int xCenter;
      private int yCenter;

```

You previously computed these values by dividing the width and height of the form by 2 (dividing in half), and that is how you'll compute the initial values for xCenter and yCenter as well, as you'll see in the new code in the constructor, below.

You'll add a new static value for the radius of the date string and add the static value Offset, discussed above:

```

C# private static int DateRadius = 600;
      private static int Offset = 0;

```

Because you want to use the same font in many places, make the font a member variable of the ClockFace class:

```
C# Font font = new Font("Arial", 40);
```

Finally, give your ClockFace class an instance of the nested class StringDraw:

```
C# private StringDraw sdToday;
```

```
VB Private xCenter As Integer  
Private yCenter As Integer  
Private Shared DateRadius As Integer = 600  
Private Shared offset As Integer = 0  
Private myFont As New font("Arial", 40)  
Private sdToday As StringDraw
```

Modifying the constructor. You are now ready to implement the changes to the ClockFace constructor. You will instantiate the StringDraw object by passing in two parameters: a string representing the current date and a reference to the current ClockFace object:

```
C# sdToday = new StringDraw(today, this);
```

```
VB sdToday = New StringDraw(today, Me)
```

You create the today string by getting the current date from the System.DateTime.Now property, calling the ToLongDateString() method.

```
C# string today = System.DateTime.Now.ToLongDateString();
```

```
VB Dim today As String = System.DateTime.Now.ToLongDateString()
```

For aesthetic reasons, remove commas from this string by calling the Replace() method of String:

```
C# today = " " + today.Replace(",", "");
```

The only other changes in the constructor initialize the current time and the x,y coordinates:

```
C# currentTime = DateTime.Now;  
xCenter = Width / 2;  
yCenter = Height / 2;
```

Resetting the center

You want the user to be able to move the clock by clicking on the form. Use the xCenter and yCenter member variables to change the center of the clock, in response to a mousedown. The event handler will readjust the xCenter and yCenter to the

values returned by the X and Y properties of the MouseEventArgs object passed in to the handler:

```
C# private void ClockFace_MouseDown(
    object sender, System.Windows.Forms.MouseEventArgs e)
{
    xCenter = e.X;
    yCenter = e.Y;
}
```

Once this is done, call `Invalidate()` to force a call to `Paint()`:

```
C#     this.Invalidate();
    }
```

```
VB Private Sub ClockFace_MouseDown( _
    ByVal sender As Object, _
    ByVal e As System.Windows.Forms.MouseEventArgs) _
    Handles MyBase.MouseDown
    xCenter = e.X
    yCenter = e.Y
    Me.Invalidate()
End Sub 'ClockFace_MouseDown
```