

## The *Paint* event

"Painting your window" means to make its appearance correct: it should reflect the current data associated with that window, and any text or images or controls it contains should appear correctly.

A window may need painting

- when data changes
- when the window has been covered and then uncovered by another window
- when the user resizes it, or scrolls its contents
- when the active application has changed

Windows sends your window a `WM_PAINT` message when it needs repainting. If you were programming directly in the Win32 API, you would write code in your window procedure to handle this message. When programming with the Foundation Class Libraries (FCL), the code in *Application.Run* contains the "message pump". When it receives a `WM_PAINT`, it constructs a *PaintEventArgs* object and calls all the methods that are registered as handlers for that event, passing the correct *sender* parameter and the newly constructed *PaintEventArgs* object.

As we saw in your first Windows program, Visual Studio even writes a skeleton event-handler for you, like this:

```
private void Form1_Paint(object sender,
System.Windows.Forms.PaintEventArgs e)
{
}
```

Normally you will ignore the *sender* parameter. (You would only need it in case more than one form used the same handler for the *Paint* event, which won't happen if you use Visual Studio. It could happen if you were writing your own code.) All the data you'll need is in the *PaintEventArgs* object.

## The *Graphics* class.

The methods that draw lines, circles, rectangles, and text in the client area of a window all belong to the *Graphics* class. You've already used one of

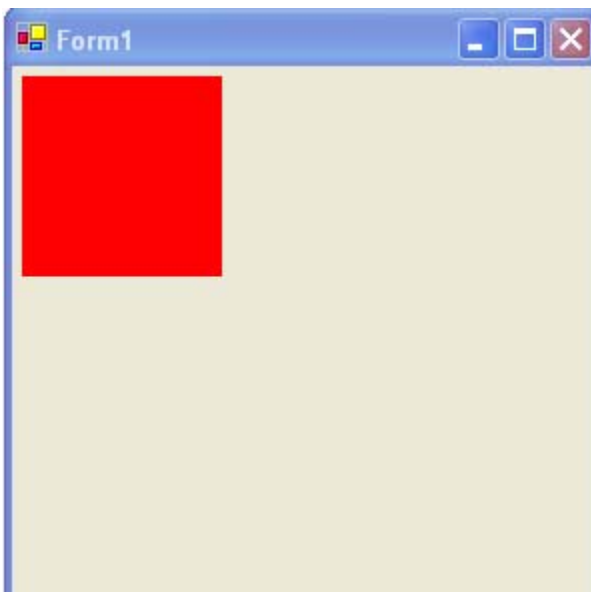
them in the call to the function `e.Graphics.DrawString` in your first program. Here `e.Graphics` is an object of type `Graphics`, and `DrawString` is a method in the `Graphics` class.

All output to the screen in Windows, except the contents of controls such as edit boxes, text boxes, and list boxes, is graphics. This includes text, which is just another kind of graphics.

Unlike the graphics systems used in either the Windows API or MFC, the system used in the FCL is (more or less) *stateless*. In practice what that means is that each call to the methods in the `Graphics` class has to have some parameters passed to it. For example, `DrawString` needs at least a font and a brush and coordinates at which to write text. This font and brush will not be “remembered” (that would require “state”), and your next call to `DrawString` will *also* need to mention a font and brush, even if they are the same as before. Here’s an example that draws a red rectangle:

```
private void Form1_Paint(object sender,
System.Windows.Forms.PaintEventArgs e)
{ Brush b = new SolidBrush(Color.Red);
  e.Graphics.FillRectangle(b,5,5,100,100); // brush, left,
top, width, height
}
```

Here’s the output:



## Client Coordinates

The coordinates used by functions in the *Graphics* class are *client coordinates*. The origin is at (0,0) in the upper left corner of the client area of the window. X increases to the right just like in mathematics, but Y increases in the downwards direction, the opposite of the usual convention in mathematics. The unit is pixels. Most monitors nowadays have “square pixels”, so that the above code draws a square (since the width and height are given by the same number). Many printers, however, do not have square pixels, and their pixels are smaller than those on a monitor. For now, we’re not printing anyway, so pixel coordinates will be fine.

## Rectangles and *FillRectangle*

Here’s an example of the use of the *Rectangle* class. This code produces the same red rectangle as the previous code.

```
private void Form1_Paint(object sender,
System.Windows.Forms.PaintEventArgs e)
{ Brush b = new SolidBrush(Color.Red);
  Rectangle r = new Rectangle(5,5,100,100); left, top,
width, height
  e.Graphics.FillRectangle(b,r);
}
```

The *Rectangle* class has two *int* fields, *X* and *Y*, that specify one corner of the rectangle. It also has two fields, *Width* and *Height*, that specify (guess what) the width and height of the rectangle. These fields can be used to get and/or set the properties of a rectangle. They are the fields used in the constructor shown in the code above. *Rectangle* also has fields *Left*, *Right*, *Top*, and *Bottom*. These fields are read-only. Check that after constructing the rectangle as above, you can insert

```
r.X = 50;
r.Y = 50;
```

and the program will build, and the rectangle is changed. On the other hand if you try

```
r.Left = 50;
```

```
r.Top = 50;
```

you will get an error message, because those properties are read-only.

Try setting `r.Width = -50`. This compiles, but `FillRectangle` produces no output.

There is also a class `RectangleF`, whose members have the same names as `Rectangle`, but their types are `float` instead of `int`. You might want to use `RectangleF` when you have set up some kind of coordinate system other than the default coordinates.

### ***Point and Size***

There is a class `Point` with two `int` members `X` and `Y`. There is also a class `Size` with two `int` members `Width` and `Height`. A rectangle can be constructed from a point and size, as well as from four ints. The following code produces the same red rectangle as before:

```
Brush b = new SolidBrush(Color.Red);
Point p = new Point(5,5);
Size s = new Size(100,100);
Rectangle r = new Rectangle(p,s);
e.Graphics.FillRectangle(b,r);
```

You might wonder why `Size` is needed; it's formally exactly the same as `Point`, i.e. two integers. The reason is that the systematic use of `Point` and `Size` should cut down on bugs. When you specify a rectangle, you can think of it in two ways: *left, top, right, bottom*, or *left, top, width, height*. Many bugs have been caused in the past by putting a number that really measures a width in for the *right* of a rectangle, or vice-versa. The way `Rectangle` is defined in the FCL, `Right` and `Bottom` are read-only; and the systematic use of `Point` and `Size` should significantly reduce the number of such errors. There is a constructor for `Point` that takes a `Size`, and vice-versa, but you shouldn't ever need them, if you've declared your variables in accordance with their intended use.

***Example 1:*** What is the center point of a `Rectangle r`?

```
Point p;
p.x = (r.Right + r.Left)/2;    // note, plus not minus here
p.y = (r.Bottom + r.Top)/2;
```

**Example 2:** What is the center point of a *Rectangle* *r*? (another solution)

```
Point p;  
p.x = r.Left + r.Width/2;  
p.y = r.Bottom + r.Height/2;
```

You can take the intersection of two rectangles:

```
r.Intersect(r2);
```

Sets *r* to the intersection of *r* and *r2*.

```
r.Union(r1,r2)
```

Sets *r* to the smallest rectangle containing both *r1* and *r2*.

```
r.Offset(30,40);
```

Adds 30 to *X* and 40 to *Y* without changing *Width* and *Height*.

**Example 3:** Is point *p* inside rectangle *r*? This is very useful for testing where the user has clicked the mouse.

```
if( r.Contains(p) )  
    { // yes, it's inside  
    }
```

An important thing to know about *Contains* is that it considers the left and top border of the rectangle to be inside the rectangle, but the bottom and right borders are outside the rectangle. This is useful when you have a grid of rectangles, like a checkerboard; then each point is in exactly one square of the checkerboard. But other times it is inconvenient, and you have to watch out for problems this “feature” may cause you.

## **ClientRectangle**

You often want to know the “client rectangle” of your window, that is, the rectangle in which your program can draw. (This excludes the title bar, menu bar, toolbar, status bar(s), and border of your window, if it has those things.) There is a *ClientRectangle* member in the *Form* class, so (by virtue of the *using* commands at the top of your file) you can just refer to *ClientRectangle* anywhere in your painting code.

**Example 4:** Center the red rectangle in the window.

```
private void Form1_Paint(object sender,
System.Windows.Forms.PaintEventArgs e)
    {   Brush b = new SolidBrush(Color.Red);
        Point p = new Point(5,5);
        Point center = new
Point((ClientRectangle.Right+ClientRectangle.Left)/2,
                                           (ClientRectangle.Bottom +
ClientRectangle.Top)/2);
        Size s = new Size(100,100);
        Rectangle r = new Rectangle(center,s);
        r.Offset(-s.Width/2,-s.Height/2);
        e.Graphics.FillRectangle(b,r);
    }
```

Observe, however, that this program does not work properly when the window is resized, or maximized and restored. This can be cured by putting this line in the form's constructor:

```
ResizeRedraw = true;
```

This causes the value of the *ClientRectangle* member to be reset, and the entire client rectangle to be redrawn, when the window is resized.

### ***DrawRectangle***

*DrawRectangle* outlines a rectangle using a pen, instead of a brush. Here's a sample call:

```
{ Pen q = new Pen(Color.Green,2); // the pen is 2 pixels
wide
    Rectangle r = new Rectangle(5,5,100,100);
    e.Graphics.DrawRectangle(q,r);
}
```

**Example 5:** Call *DrawRectangle* on the entire client rectangle, with a pen of width 1. You will only see the top and left sides drawn. *ClientRectangle* is set so that every point in the client area is inside it, according to *Contains*. But the right and bottom edge are outside. Hence, they aren't visible. If you wanted to see them you would have to create another rectangle with a slightly smaller *Width* and *Height*.

## *DrawEllipse*

*DrawEllipse* draws an ellipse bounded by a specified rectangle.

**Example 6:** a circle is a special case of an ellipse, when the bounding rectangle is a square.

```
Pen g = new Pen(Color.Green,2); // the pen is 2 pixels wide
Rectangle r = new Rectangle(5,5,100,100);
DrawEllipse(g,r);
DrawEllipse(g,5,5,100,100); // this works too
```

## *FillEllipse*

*FillEllipse* fills an ellipse bounded by a specified rectangle. You call it like *DrawEllipse*, but with a brush instead of a pen. It fills a slightly smaller ellipse than drawn by *DrawEllipse*, so you can fill and outline an ellipse by calling *FillEllipse* and then *DrawEllipse*. See Petzold, p. 204, for more details.

## **Video Memory**

The graphics board (or "card") contains some memory that is used to update the screen. We can call this "video memory". 60 to 75 times a second this data will be used to drive the electron gun that makes the screen's pixels light up. [Television updates the screen 30 times a second, but computer monitors go twice as fast.] Thus, writing or drawing to the screen is done by changing some bits in video memory.

[Some diagrams were shown on the board which are not in these slides, but the slides are still (hopefully) complete.]

When a function like *DrawString* or *FillRectangle* is executed, the result is to change certain bits in video memory (on the graphics card). The next time the screen is refreshed these bits will result in the appearance of text or a colored rectangle on the screen. The step from the modification of video memory to the appearance on the screen is managed by the hardware (and/or firmware) on the graphics card, not by your (compiled) code. The details of the modification of video memory are also not handled by your code, but by the "driver" for the graphics card. The Windows implementation of

*DrawString*, for example, calls the appropriate driver supplied by the manufacturer of the graphics card.

But, to cope properly with problems of flicker, etc., you must understand how this works: your code modifies video memory.

## **Clipping**

Drawing to the screen takes time, sometimes too much time. So it is useful to be able to "clip" output to a given rectangle. Only pixels within that rectangle are changed. This is called "clipping". Modern devices support clipping in hardware, for speed.

## **The invalid rectangle**

Windows keeps track of a rectangle which includes all areas of the window which will need modification at the next WM\_PAINT message. This rectangle is called the "invalid rectangle". Note, each window has its own invalid rectangle.

Just before *Form1\_Paint* is called, the clipping rectangle will be set (by Windows) to what was the invalid rectangle up until *Form1\_Paint* was called. Then the invalid rectangle will be set to the empty rectangle again. You will be able to find out, if you need to know, what the clipping rectangle is, by consulting *e.ClipRectangle*, where *e* is the *PaintEventArgs* object passed to your *Form1\_Paint*.

In most Windows programs, you will not need to know or take explicit account of *ClipRectangle*. Instead, you write code in your *Form1\_Paint* as if the whole window had to be repainted. Clipping will automatically prevent your program from inefficiently repainting parts of the screen that do not need it.

The only time you would have to worry about repainting less than the whole window is in the rare case that it costs a lot of time to compute *what* to paint, e.g. in certain mathematical computer graphics programs.

### ***Invalidate()***

When the document changes you do NOT immediately paint the screen. Instead, you call the *Invalidate* member function of your form. This causes a WM\_PAINT message to be sent to the window, and then *Form1\_Paint* will be called. The rule is, there is only one place in which you can paint the screen: in *Form1\_Paint*. As a very expert Windows programmer, you might possibly violate that rule in certain very special cases, but not in this course!

The FCL helps you with this because you need a *Graphics* object to draw, and you get one only as a member of the *PaintEventArgs* parameter to *Form1\_Paint*.

### ***Invalidate(Rectangle r)***

*Invalidate* makes the invalid rectangle the entire client area of your window. If it is called often, this can produce flicker as your window is redrawn many times in succession. In such situations, it is possible to "invalidate" only a specified rectangle, containing the portion of the window you need redrawn, rather than the entire window. You pass a rectangle parameter to *Invalidate* to do this. You need this when programming animations or when drawing in response to the mouse.

