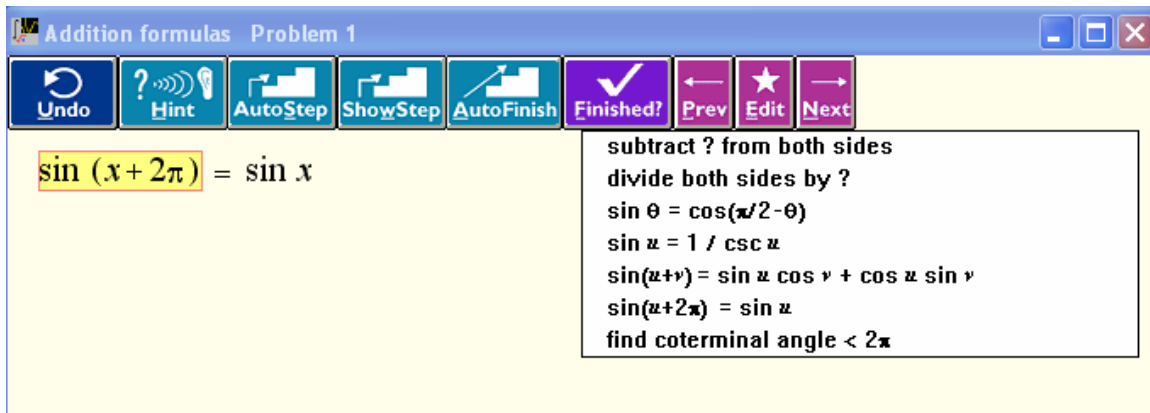


## Owner-Draw Menus

Normal menus are always drawn in the same font and the same size. But sometimes, this may not be enough for your purposes. For example, here is a screen shot from MathXpert:



Notice in the menu that mathematical variables appear in italic type, while function names and parentheses appear in Roman type, as is the custom in printed mathematics; and a couple of Greek letters are also used.

In .NET 2005, it's easier to take control of the appearance of your menu than it was in any previous way of programming Windows. Namely: you can just handle a *Paint* event for the menu item in question. First, set the *DisplayStyle* property of the menu item in question to *None*. Then experiment to see how much of your menu item's functionality is left. You'll see that while the text doesn't appear (even if the *Text* property is set), you can still see the background change when the item is selected, and the default background of blue where the image could go is still there.

The background is being drawn by the *DrawBackground* method. That takes care of the highlighting of selected items. You can override that method if you want to draw the backgrounds yourself—but I can't imagine why you would want to do that. (In the Windows API and MFC, you *had to* draw your own background if you wanted to draw the menu item.)

You can use any graphics methods in drawing your item. We will work through a simple example that presents two menu items using more than one font.

Now it only remains to supply the code. Begin by creating three fonts:

```
static Font RomanMenuFont = SystemInformation.MenuFont;
static Font ItalicMenuFont = new Font(RomanMenuFont,
                                     FontStyle.Italic);
static Font greekFont = new Font("Symbol", 9);
```

*Why did I make these static?* That way, they are created and destroyed only once, at the beginning and end of the program's lifetime. One could also create the fonts new every time the event handlers are called; that also works. Fonts (like pens and brushes) are probably just small data structures and the actual glyph information is probably not copied every time you create a font, but that source code is proprietary, so we don't know for sure. In any case, it seems sensible to me that if you are going to use a font (or pen or brush) many times you should make it static, while if it's going to be rarely used, you should make it when you need it.

When you type this code in, look at all the possible fields of *SystemInformation*. Since the fonts used for menus can vary from user to user, to make our menus look as much as possible like normal menus, we choose this way of defining our fonts.

We still have to worry about the color for the menu text. We could choose our own colors, but that would not be a good idea. The colors may vary from one version of Windows to another, and besides, they are user-configurable. Unless we purposely want to make our menus have a different color from normal menus, we should use the *SystemBrushes* class as shown here. (More explanations of the code come after the code.)

```
private void MyToolStripMenuItem_Paint(object sender,
PaintEventArgs e)
{
    Graphics g = e.Graphics;
    ToolStripMenuItem mi = (ToolStripMenuItem)sender;
    Brush b;
    if(mi.Selected)
        b = SystemBrushes.MenuHighlight;
    else
        b = SystemBrushes.MenuText;
    // or SystemBrushes.FromSystemColor(SystemColors.MenuText);
    Rectangle r = ClientRectangle;
    r.X += SystemInformation.MenuButtonSize.Width;
    drawMath(mi.Text, b, g, r);
}
```

The last two lines of the code still require an explanation: It may seem that I should just write

```
drawMath(mi.Text,b,g,ClientRectangle);
```

But there is a place at the left of the menu for an image or a check mark. We need to leave space at the left for that. Since *ClientRectangle* is read-only, we create another rectangle *r* for that purpose.

## drawMath

To start with, you can just use *DrawString* for *drawMath*, but here's a somewhat more illustrative piece of code:

```
private String translate(String x)
// x should be "alpha" or "theta"
// return the corresponding Greek character as a string
{ String alpha = "" + (char) 0x61;
  String theta = "" + (char) 0x71;
  if (x == "alpha") return alpha;
  if (x == "theta") return theta;
  throw new Exception("Unsupported greek letter");
}

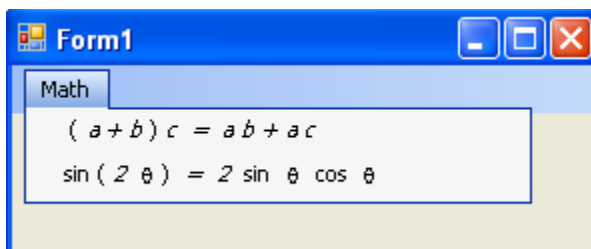
private void drawMath(String X, Brush b, Graphics g, Rectangle r)
// handle two kinds of backslash escapes:
// \sin, \cos, etc. to print function names in Roman type
// \alpha, \beta, etc. to print Greek letters
// use Roman type for parentheses and italic for variables
{
  int n = X.Length;
  int i;
  float x = 0;
  String t;
  Font f;
  float fudge = 2; // to adjust height of menu item in its rectangle
  for(i=0;i<n;i++)
  { char c = X[i];
    if(c == '(' || c == ')')
    { t = "" + c;
      f = romanFont;
    }
    else if (c == '\\')
    {
      String s = X.Substring(i + 1);
      c = s[0];
      if (s.Substring(0, 5) == "alpha" ||
          s.Substring(0, 5) == "theta"
```

```

    )
    {
        t = translate(s.Substring(0, 5));
        f = greekFont;
        i += 5;
    }
    else if (s.Substring(0, 3) == "sin" ||
            s.Substring(0, 3) == "cos" ||
            s.Substring(0, 3) == "tan"
            )
    {
        t = s.Substring(0, 3);
        f = romanFont;
        i += 3;
    }
    else
        throw (new Exception("Unsupported backslash escape"));
} // end of backslash escape handling
else // a variable (not Greek)
{
    t = " " + X[i];
    f = italicFont;
}
// OK, now we've got the string and font, draw!
g.DrawString(t,f,b,r.X + x, r.Y + fudge);
x += g.MeasureString(t, f).Width;
}
}

```

Here's a screen shot showing the result:



Now you see that the spacing you got from *MeasureString* is far from ideal. And the “2” should be in Roman type, not italic. It’s a long ways from here to really nice-looking mathematics, but the rest is just sweat—you’ve learned the relevant principles of Windows programming.

### Getting rid of the image margin

If you would like to get rid of the “image margin” entirely, it doesn’t seem to be easy. The *ContextMenuStrip* class has a property *ShowImageMargin*

that you can set to false, but the *MenuStrip* class doesn't have that property. It turns out that the *ToolStripDropDownMenu* class does have that property. So you need to find the menu item on your menu bar whose drop-down menu contains the item(s) for which you don't want an image margin. (Apparently you must have it for all or none of the items on a drop-down menu.) Here's the code that does it:

```
((ToolStripDropDownMenu)(mathToolStripMenuItem.DropDown)).ShowImageMargin = false;
```

## Font Choice Issues

One final issue about owner-draw menus is this: it often does not look good to mix owner-draw menu items with regular menu items. As mentioned above, the system font can differ from system to system, and you can't be sure that your beautifully designed owner-draw menu items will look good in some other font. Therefore, you might want to stick to the font you designed them with. But then, it may look strange if other items on the menu are in the normal menu font. Also, if your menus are dynamically created (as in MathXpert) it becomes a bookkeeping nightmare to keep track of which items are owner-draw and which ones are not. After wrestling with this for a long time, finally I just made *all* the menus in MathXpert owner-draw.

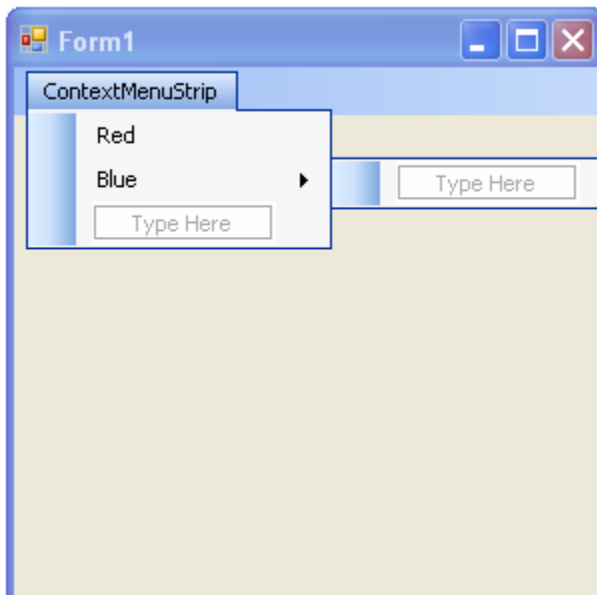
## Context Menus

A *context menu* is a menu that is not attached to a menu bar or another menu item, but comes up independently. The usual custom in modern program design is that context menus are raised when a *MouseUp* event occurs from the right mouse button. We often say, on a "right click", but if you'll look carefully, most programs wait until the *MouseUp*.

The FCL automates the display of a context menu for you. Each *Form* (or class derived from *Form*) has a *ContextMenuStrip* property. You can construct a *ContextMenuStrip* from an array of menu items, and use line of code like

```
this.ContextMenuStrip = new  
ContextMenuStrip(myArrayOfMenuItems);
```

However, Visual Studio's form designer saves you from taking even this much trouble. From the Toolbox you can drag and drop a *ContextMenuStrip* onto your form. It will appear at the bottom of the designer window. When you select it, it will appear in the form editor temporarily so that you can design or edit it:



So far, nothing connects this context menu to your form. You could add more than one context menu in this way—which one will come up on right-click? To specify that, you must set the *ContextMenuStrip* property of *Form1* to *contextMenuStrip1*.

To show how this works, I added three items, *Red*, *Blue*, and *Yellow* to this context menu, and gave them all the same *Click* handler, *ContextMenuHandler*. (Type that out for the first item, then select it for the rest.) The following code just changes the background color of the form:

```
private void ContextMenuHandler(object sender, System.EventArgs e)
{
    ToolStripMenuItem mi = sender as ToolStripMenuItem;
    if(mi != null)
    {
        Color c;
        if(mi.Text == "Red")
            c = Color.Red;
        else if (mi.Text == "Blue")
            c = Color.Blue;
        else
            c = Color.Yellow;
    }
}
```

```
        this.BackColor = c;  
        Invalidate();  
    }  
}
```

You perhaps haven't seen the *sender as ToolStripMenuItem* construction in C# before. This is more or less equivalent to casting *sender* to *ToolStripMenuItem*, except that if *sender* really isn't a *ToolStripMenuItem*, the behaviour is different. An explicit cast will throw an exception. The *as* construct will simply make *mi* equal *null*. Thus if some other idiot on my programming team calls this code with a non-*ToolStripMenuItem* argument, nothing will happen—no exception will be thrown. That's the reason to use an *as* construct instead of a cast.

At this point, the context menu should be working. If it doesn't, check for these common mistakes: Did you remember to set the *ContextMenu* property of *Form1*? Did you remember to set the *Click* handlers of all the menu items on the context menu to *ContextMenuHandler*?

This example, however, doesn't illustrate the full possibilities of a context menu. After all, we haven't made much use of the *context*, that is, of the point where the mouse was clicked. In this example, any point in the client area of the form brings up the same menu. But think, for example, of the *Sudoku* program. We might want to bring up one context menu on a square that already contains a number, and another context menu on a square that doesn't already contain a number, and no context menu at all on a square that contains a number that was part of the original problem (and hence can't be changed). How would we program that?

In that case, we would *not* set the *ContextMenuStrip* property of *Form1*. We would write our own handler for *MouseUp*, and in that handler, we would use the *Show* method of the *ContextMenu* class. To demonstrate how this works, we need two different context menus. Drag and drop *contextMenu2* onto your form, and edit it to have two menu items, *Circle* and *Square*. Name your menu items *circleItem* and *squareItem*, and give them both the handler *ContextMenu2Handler*. The plan is this: the program will draw either a filled square or a filled circle. Right-clicking inside the square will bring up one context menu, and right-clicking outside the square will bring up the other context menu. Here's how to code this:

First, you need some member variables:

```
private Rectangle m_theRect = new Rectangle(10,10,50,50);
private Color m_theColor = Color.Red;
private int m_theShape=0; //0 means square, 1 means circle
```

Next, be sure you have removed the *ContextMenuStrip* property of *Form1*.  
Write a *Paint* handler that draws the square or circle:

```
private void Form1_Paint(object sender,
System.Windows.Forms.PaintEventArgs e)
{
    Brush b = new SolidBrush(m_theColor);
    if(m_theShape == 0)
        e.Graphics.FillRectangle(b,m_theRect);
    else
        e.Graphics.FillEllipse(b,m_theRect);
}
```

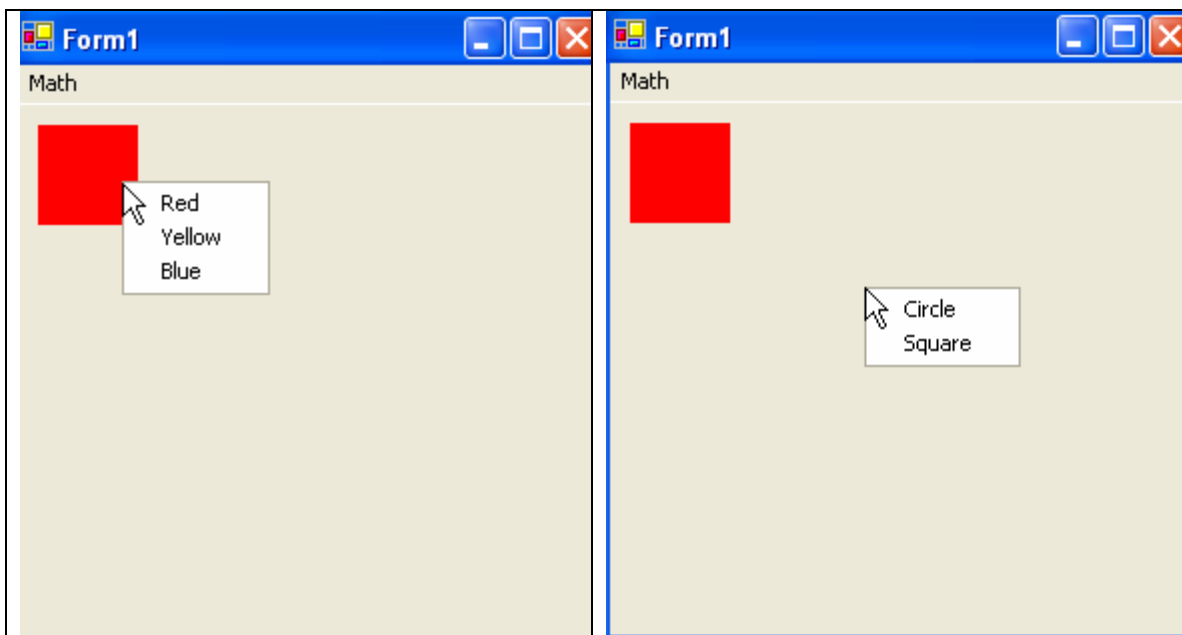
Here is the handler for the new context menu:

```
private void ContextMenu2Handler(object sender,
System.EventArgs e)
{
    ToolStripMenuItem mi = sender as ToolStripMenuItem;
    if(mi != null)
    {
        if(mi == circleItem)
            m_theShape = 1;
        else
            m_theShape = 0;
        Invalidate();
    }
}
```

and here is the heart of the exercise, the code that brings up a different context menu depending on where you right-click:

```
private void Form1_MouseUp(object sender,
System.Windows.Forms.MouseEventArgs e)
{
    if(m_theRect.Contains(e.X,e.Y))
        contextMenu1.Show(this,new Point(e.X,e.Y));
    else
        contextMenu2.Show(this,new Point(e.X,e.Y));
}
```

The following two screen shots show the different context menus:



You will see from the screen shots that the point passed to *contextMenu1.Show* determines the upper left corner of the context menu. You could adjust that in your code. Try clicking near the right-hand edge of the window. Will the context menu stick out of the window or be clipped? [It will stick out]. Now move the window to the right-hand edge of the screen, so the context menu can't possibly stick out. Now what will happen? Will the context menu be displayed or clipped? [It's displayed in a different place, shifted left so it will all be visible].

*Question:* Can you make the context menu items have a different effect depending on where the menu was brought up? For example, could we use the color context menu to change the color of the rectangle if the user right-clicks in the rectangle, and the background color if the user clicks on the background?

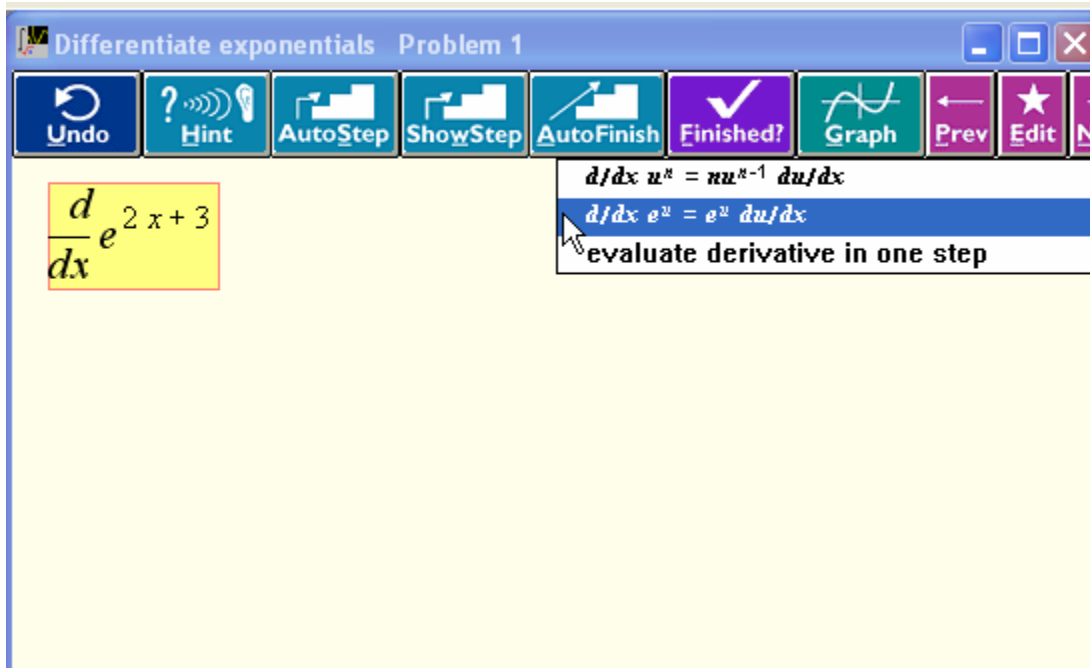
*Answer:* Yes, but not cleanly. In the *MouseUp* event you have access to the point, but you would have to store it in a static variable so that it would be accessible in the menu item handler. Alternately, if you knew in advance that there would be only one square or circle, you could just use two

different menus with identical text. That wouldn't work if the user were allowed to create new squares and circles.

*Remark:* Note that *this* is passed as the first argument to the *Show* command that brings up the menu. That makes the coordinates be interpreted as *Form1* coordinates. If you only pass the coordinates, they will be interpreted as screen coordinates. The first position can contain the name of any control; since technically *Form1* is a control (remember *Form* is derived from *Control*), passing *this* works.

## A Big Improvement over the Win32 API

Here's a problem that drove me crazy when programming *MathXpert*. Look at this screen shot:



The user has just pressed *ShowStep* and the program showed the user what expression they should select, and then the context menu came up as a result of that selection, and the cursor moved under program control to select the menu item that the user is advised to choose. The programming problem is this: how do you get a menu to come up with a specified item pre-selected?

I never was able to find a way to do this in the Win32 API. In the end, I completely reprogrammed context menus from scratch, in order to achieve this result! It was vital to implement my design that the program should be able to show the user a good way to take the next step, not just to take it for them.

But in .NET, at least in the 2005 version, it's easy. The *ToolStripMenuItem* class has a *Select* method. All you have to do is call the *Select* method of the Menu item you want highlighted. Here's an example:

```
blueToolStripMenuItem.Select();  
contextMenuStrip1.Show(this, e.X, e.Y);
```

Here's the result:

