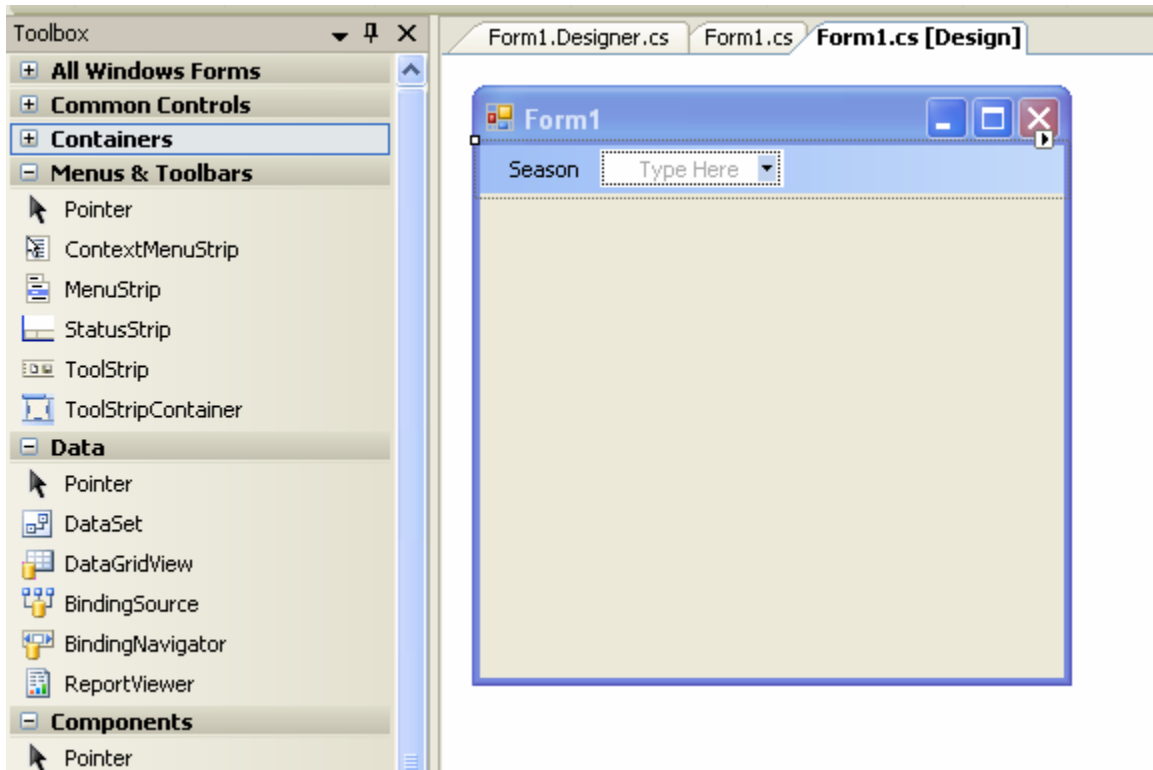
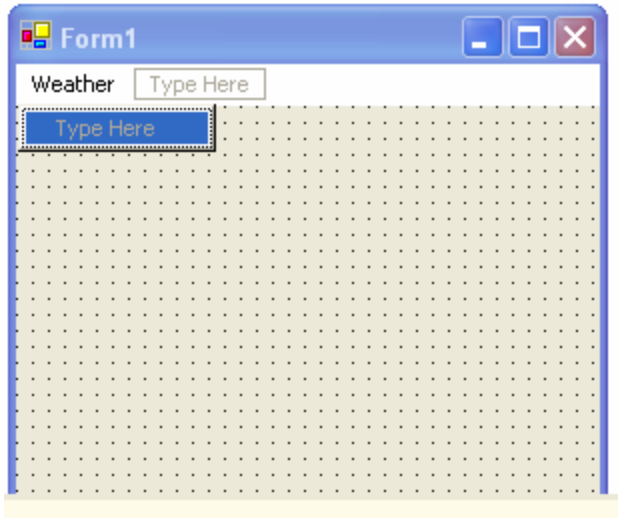


Menus

In .NET, a menu is just another object that you can add to your form. You can add objects to your form by drop-and-drag from the *Toolbox*. If you don't see the toolbox, choose *View / Toolbox* in the main menu of Visual Studio. You should see this:



You'll find *MenuStrip* listed in the Toolbox. Drag one to your form. Where it says *Type Here*, type *Weather*. Then you'll see this:



Now you see two *Type Here* boxes. One permits you to add a new item to the menu bar; the other permits you to add more menu items on the *Weather* menu. Add four items there: *Sun, Rain, Clouds, Snow*.

Build and run the program. Observe that the menu is there, but the items don't do anything. That shouldn't be a surprise, because we haven't added handlers for them yet. Also, we should change the name of the program to *Menu Demo* instead of *Form1*.

What we want to do in this simple demo program is just have each menu item cause the display of a different string. To arrange that, we want to add a member variable *m_theString* to the *Form1* class. Then the menu handlers only have to change the value of that variable (well, almost "only").

To add a member variable, the easiest way is just to type the code yourself.

```
private String m_theString = "Sun";
```

Type this in right after the existing member variables in your *MenuDemo* class. Build your program to check that it compiles. Now, try it again with a lower-case *s* in *string*. That compiles too! Wait a minute, C# is case-sensitive, isn't it? What's going on here?

Answer: *string* is a C# data type, and *String* is one of the .NET Foundation Classes. Because your file has *using System;* at the top of the file, we don't have to say *System.String*.

OK, that accounts for why they both compile, but why do two different string classes exist? Or more usefully, what are the differences between them? The documentation for *String* tells us that the elements of a *String* are of type *System.Char* (occupying one byte, it seems) while the elements of a *string* are Unicode characters (occupying two bytes). However, a *String* can contain Unicode characters—it's just that sometimes two *System.Char* objects will be used to represent them. Let's leave that aside for now, and just use *String* objects until further notice.

Adding Menu Handlers

Note that each menu item is an object; you can right click it in Design View and see its properties. It has a *Text* property, which is what is displayed, and it has a *Name*. Visual Studio has named them things like *sunToolStripMenuItem*. You can change that on the property sheet of each menu item, but *only* before you have added a handler, so if you want to change them, now is the time. The names Visual Studio gave them are constructed from the text you typed, so they're probably OK.

To add the handlers for these menu items, just double click the menu items in the form editor. Make your four handlers look like this:

```
private void Sun_Click(object sender, System.EventArgs e)
{
    m_theString = "Sun";
    Invalidate(); // this line will be explained below
}
```

Run the program and check that it works as hoped.

Invalidate

Now comment out the *Invalidate()* line in the *Rain* handler, and see what happens. When you choose *Rain*, the string no longer changes on the screen. It *does* change in memory though, as you can see by resizing the window to obscure the string *Sun* and then making it bigger again; now *Rain* will show correctly! The reason it didn't show before is that no *Paint* event was generated, so your *Form1_Paint* code was never called. When you resized the window, a *Paint* event was generated by the system, and your *Form1_Paint* code was called then. The purpose of the *Invalidate()*

call is to generate a *Paint* event that will cause the entire client area of the program to be repainted when it is processed.

Draw only in processing a *Paint* event.

This is a fundamental principle of Windows programming, with very few exceptions that occur only in advanced and special situations. The normal way of programming is this:

- handlers of other events change data (member variables) and then call *Invalidate*.
- The handler of the *Paint* event changes the appearance of the window.

The reason for this is that the *Paint* method has to be able to correctly present the window whenever it needs to be painted—when it is resized, or when it appears after another window has been painted over it, as well as when the user changes the data using menus or other controls.

The Code to Create Menus

Study the code in *InitializeComponent* that Visual Studio wrote when you used the menu editor. You could have written that code yourself. Compare the code for menus in the Petzold book. Petzold offers a criticism of the code Visual Studio writes for menus: it introduces a new member variable for each menu item. Most Windows programmers are glad to let Visual Studio write menu code for them, but sometime you will want to do something with menus that Visual Studio can't do for you, and therefore you must learn something about how the code works. In fact, later in this lecture you'll see a place where the Visual Studio code seems awkward.

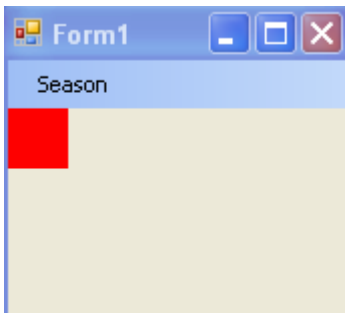
A little history

In earlier days, menus were regarded as a part of every Windows program. But starting with the 2005 version of .NET, menus are considered “just another control”, like buttons or combo boxes. Their properties can be set so the user can change their location, for example. This means that now, the menu occupies part of the client area. Formerly, the menu was (like the title bar) *not* considered part of the client area. That means that your *Paint* code will have to take account of the height of the menu bar when it

calculates coordinates for drawing things. For example, just centering something in the window will get more complicated, if you want to center it in the rectangle that remains below the menu bar. Here is an example of code to draw a small red rectangle just below the menu bar:

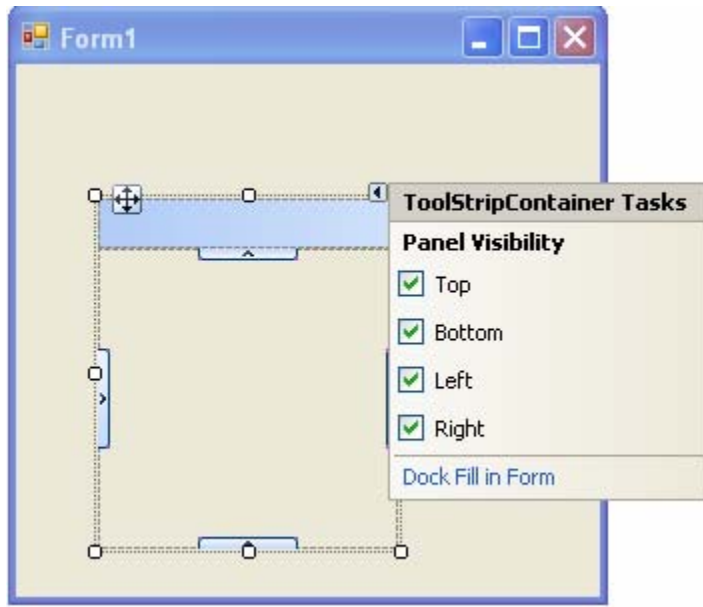
```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    int y = menuStrip1.Height;
    Rectangle r = new Rectangle(0, y, 30, 30);
    e.Graphics.FillRectangle(Brushes.Red, r);
}
```

and here is the result:



ToolStripContainer

The problem just discussed can be remedied as follows. When you're starting out, instead of first adding a *MenuStrip* to your form, add a *ToolStripContainer*. You'll see



Remove the checks from all but “top”, and then drag your *MenuStrip* onto the container’s top panel. Now right-click in what appears to be the main window, and choose *Properties*. Observe that you’re not in *Form1* but in a *Panel* object. What has happened is that the *ToolStripContainer* window contains five smaller windows, called “panels”. The top panel contains your menu, and the center one, with the name *ContentPanel*, is available for you to draw in, as you formerly drew in *Form1*. Now, you won’t have to adjust coordinates for the height of the menu.

Keyboard Shortcuts

Menu items in Windows often have one letter underlined. That indicates that you can invoke that menu item from the keyboard using the Alt key with the underlined letter. These keyboard shortcuts are not set up automatically for you, even by Visual Studio, because it’s your responsibility to avoid duplicates. For example, in the *Weather* program, both *Snow* and *Sun* begin with *S*, so we can’t use the first letter of each item as a shortcut key. You indicate a shortcut by putting & before the shortcut key in the Text property of the menu item, for example “&Sun” and “S&now”. You can do that, as usual, either directly in the initialization code or using the property sheet of the menu items. This scheme of using ampersand to specify Alt-key shortcuts is as old as Windows itself and very familiar to many Windows users.

There is another way to specify shortcuts, usually used if you want Control keys, Control-Shift keys, or function keys to be shortcuts. Two properties on the menu item's property list control this: the `Shortcut` property and the `Show Shortcut` property. You use the `Shortcut` property to select the shortcut key, and the `Show Shortcut` property to have it shown on the menu item. (Why would you ever want to have an invisible shortcut key? I can't imagine that it would ever be a sensible choice.) In my opinion, most users won't use such shortcuts anyway, and if you think you need them, you should redesign your interface instead.

Checking and Unchecking Menu Items

Let's improve the *Weather* program so that the currently selected weather has the corresponding menu item checked. We can start by setting the `Checked` property of the *Sun* menu item to true, since initially the weather is *Sun*. Now, each menu item handler has to check itself, and uncheck all the other items. You can see from *InitializeComponent* that the names Visual Studio gave to the menu items are what we specified in the `Name` property. The following code will do the job:

```
private void Rain_Click(object sender, System.EventArgs e)
{
    Sun.Checked = Clouds.Checked =
        Snow.Checked = false;
    Rain.Checked = true;
    m_theString = "Rain";
    Invalidate();
}
```

You have to put in similar code for each of the four menu handlers. Now what if we want to add a fifth kind of weather? Not only will we have to write a new menu handler, but we'll have to modify each of the existing four. This isn't ideal, and we can begin to sympathize with Petzold, who would have written just one menu handler and used it to process all these related menu items, using (*MenuItem*) `sender` to see which menu item had been clicked.

You could convert your Visual Studio-written code to use a single event handler for these menu items by modifying the lines

```
this.Sun.Click += new
System.EventHandler(this.Sun_Click);
```

It's too early in your Windows programming career to try this right now; but not too early to be exposed to the idea that you can write your own code or modify the code written for you by Visual Studio.