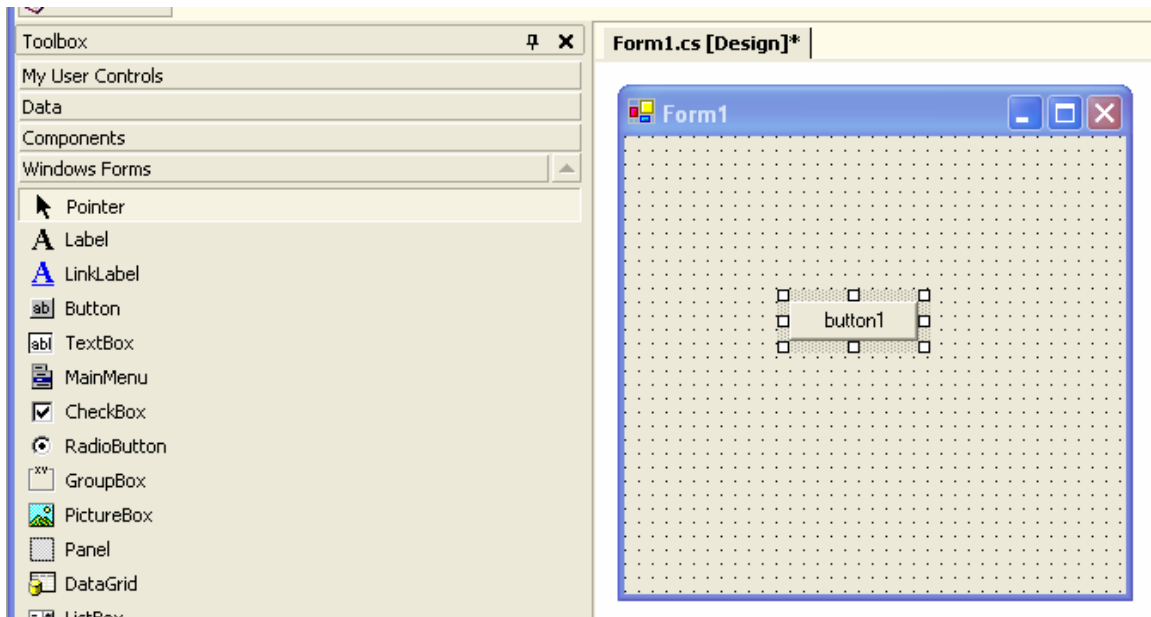


Push Buttons

It's time to start programming. Make a new project *ControlDemo*. If you don't see the *Toolbox*, make sure you have the form editor window active (not your source code window), and if that doesn't do it, choose *View / Toolbox* from the Visual Studio menu.

Now drag a Button from the Toolbox onto your form. (A Button in the tool box is a push button—there is a separate item for “radio button”.)



Using the property sheet of the button, change its *Name* property to *PushMe* and its *Text* to *Push Me*. (If you try to put a space in the *Name* property, you'll get an error message.) Change the text property of the form itself to *Control Demo*. You can build and run the program, but of course nothing happens when you push the button. Observe, though, that it does visibly “push” when you click on it. While the program is running, do this: depress the left mouse button over the button, then, holding the button down, move off the button and back on. There is an illusion that the button is being pushed and released. How is this accomplished?

Look carefully at the way the button is drawn. If you think about this, you will realize that, even though you are still a novice at Windows programming, you could program this effect using the graphics functions you know already! But, you don't have to, because the `Button` class is predefined.

Now let's see what source code the form editor wrote.

```
private System.Windows.Forms.Button PushMe;
```

So the class `Button` is derived from `Forms`. The rest of the code is in `InitializeComponent`:

```
private void InitializeComponent()  
{  
    this.PushMe = new System.Windows.Forms.Button();  
    this.SuspendLayout();  
    //  
    // PushMe  
    //  
    this.PushMe.Location = new System.Drawing.Point(96, 96);  
    this.PushMe.Name = "PushMe";  
    this.PushMe.TabIndex = 0;  
    this.PushMe.Text = "Push Me";  
    // code for Form1 omitted here  
    this.ResumeLayout(false);  
}
```

The first line creates the `PushMe` button, making it a child window of your form window,

The lines about `Location`, `Name`, and `Text` are clear enough. The line about `TabIndex` specifies the place of this control in the "tab order". In Windows, the tab key will take the user from one control to another in any window that has controls as children.

Note the two lines involving `SuspendLayout` and `ResumeLayout`? What are those about? Here is the answer to that question, clipped from the FCL documentation:

The **Layout** event occurs when child controls are added or removed, when the bounds of the control changes, and when other changes occur that can affect the layout of the control. The layout event can be suppressed using the [SuspendLayout](#) and [ResumeLayout](#) methods. Suspending layout allows for multiple actions to be performed on a control without having to perform a layout for each change. For example, if you resize and move a control, each operation would raise a **Layout** event.

Visual Studio has, in its autogenerated code, used *SuspendLayout* and *ResumeLayout* exactly in accordance with this documentation. Note that it is the layout of *Form1* that is being suspended. That might seem strange as the *PushMe* button is the “control” we’re working on, but remember that *Form* is derived from *Control*, so technically the *Form* itself is a control, in the sense of being an object of type *Control*.

Now let’s make something happen when the button is pushed. To do this, we need to have a handler for the *Click* event that is generated when the user clicks on the button. Since we don’t care *where* on the button the user clicks, the *Click* event is used instead of the *MouseDown* event. You can add a click handler just by double-clicking the button in the form editor.

This will add the handler to your *Form1* class. This is where you want to handle that event: normally events generated by one of the standard controls are handled by the parent of the control. (That’s why it seems a bit misleading that forms are technically controls, as in our forms we do handle events generated by the forms, such as *MouseDown*, etc.)

The simplest thing I could think of to have a button do is change its own color. For example:

```
private void PushMe_Click(object sender, System.EventArgs e)
```

```
{  
    PushMe.BackColor = Color.Red;  
}
```

Try it: it works—the button turns red when it is clicked.

This code is in the *Form1* class, so if we just write *BackColor = Color.Red*; it will turn the entire form red when the button is clicked.

Note: It does not *ALSO* turn the button red: you did not see a red form and a normal-colored button. If you were writing your own code for this, you would find that the button inherits certain properties, such as background and foreground colors, from its parent, unless these have been explicitly set. In Visual Studio 2008, the design editor generates code to explicitly set the foreground and background colors of controls. That makes your life easier, because you probably don't want background colors to be inherited.

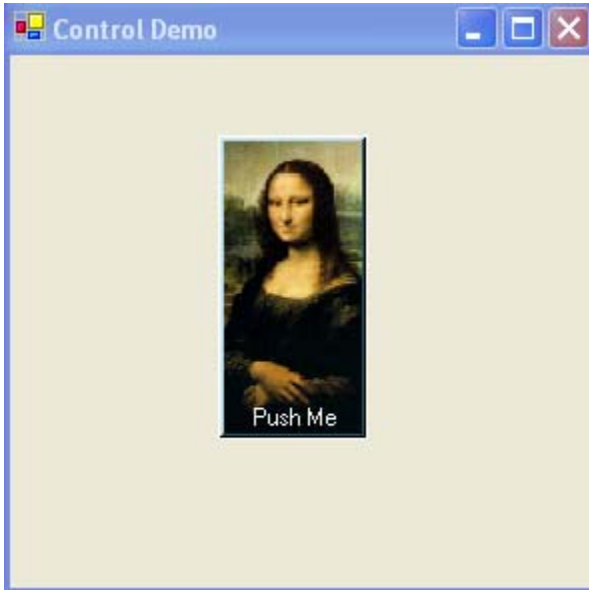
The *Anchor* property

Examine the property sheet of your button. You will see an *Anchor* property. By default this property has the value *Top,Left*. This means that when the form is resized, the button will always keep the same distance from the top and left edges of the form's client area as it had to begin with. Try the other possible values of this property. What happens if the form isn't anchored to any edge? What happens if you anchor it to all the edges?

The *Image* property and other useful properties

Before .NET, it was a relatively complicated task to put images on a button, especially in combination with text. But watch this:

On your button's property sheet, look for the *Image* property and browse to the *monalisa.bmp* file that we used before. Now the image appears in the form editor, but the button is too small, so resize the button to fit the image. The text is in an awkward position and color: fix that using the *ForeColor* and *TextAlign* properties. Here's a screen shot of the running program:



OK, so you can't get the text to an *arbitrary* position this way. That can be done in another way, but this is still pretty good for just a few clicks in the form editor.

Next, use the *Font* property to make the text appear in italic type. OK, so you still can't put a mathematical formula on a button, or mix two different fonts. That too can be done in another way, but this is still pretty good for just a few clicks in the form editor.