?Published on The O'Reilly Network (http://www.oreillynet.com/)
?http://www.oreillynet.com/pub/a/dotnet/2002/03/18/customcontrols.html
?See this if you're having trouble printing code examples

Creating Custom .NET Controls with C#

by Budi Kurniawan
03/18/2002

Windows programmers have a wide variety of controls to choose from in the `System.Windows.Forms` namespace in .NET's Framework class library. You have controls as simple as `Label`, `TextBox`, and `CheckBox`, as well as controls as sophisticated as the `MonthCalendar` and `ColorDialog` controls. These Windows controls are more than enough for most applications; however, sometimes you need controls that are not available from the standard library. In these circumstances, you have to roll up your sleeves and write your own. This article shows you how to develop a custom control with C# and presents a simple custom control.

Before you start writing the first line of code for your custom control, you should familiarize yourself with two classes in the `System.Windows.Forms` namespace: `Control` and `UserControl`. The `Control` class is important because it is the parent class of Windows visual components. Your custom class will be a descendent of the `Control` class as well. Your custom controls, however, don't normally inherit directly from the `Control` class. Instead, you extend the `UserControl` class. The first two sections of this article discuss these two classes. In the final section, you'll build your own custom control, the `RoundButton` control.

The Control Class

The `Control` class provides very basic functionality required by classes that display information to the Windows application user. This class handles user input through the keyboard and the mouse, as well as message routing and security. More importantly, the `Control` class defines the bounds of a control (its position and size), although it does not implement painting.

Windows forms controls use *ambient properties*, so child controls can appear like their surrounding environment. In this context, "ambient" means that the property is, by default, retrieved from the parent control. If the control does not have a parent and the property is not set, the control tries to determine the value of the ambient property through the `Site` property. If the control is not sited, if the site does not support ambient properties, or if the property is not set on the `AmbientProperties` object, the control uses its own default values. Typically, an ambient property represents a characteristic of a control, such as `BackColor`, that is communicated to a child control. For example, by default a button will have the same `BackColor` as its parent form.

A number of the `Control` class's properties, methods, and events are carried through by its child classes without any change.

The Control Class's Properties

The following are some of the `Control` class's most important properties:

BackColor

The background color of the control, represented by a `System.Drawing.Color` object. You can programmatically assign a `System.Drawing.Color` object to this property using code like this:
`control.BackColor = System.Drawing.Color.Red`

Enabled

A Boolean that indicates whether or not the control is enabled. The default value is `True`.

Location
The position of the top-left corner of the control in its container, represented by a `System.Drawing.Point` object.

Name
The name of the control.

Parent
Returns a reference to the container or parent of the control. For example, the parent of a control that is added to a form is the form itself; if we add `Button1` to a form, we can change the title of that form to "Thank you": `Button1.Parent.Text = "Thank you"`

Size
The size of the control, as represented by a `System.Drawing.Size` object.

Text
The string that is associated with the control. For example, in a label control, the text property is the string that appears on the label body.

The Methods of the Control Class

Some of the frequently used methods of the `Control` class are:

BringToFront
Shows the entire control, in cases where some other control is overlaying it.

CreateGraphics
Obtains the `System.Drawing.Graphics` object of the control, on which you can draw using the various methods of the `System.Drawing.Graphics` class. For instance, the following code obtains the `Graphics` object of a button control called `Button1`, and then draws a diagonal green line across the button's body:

```
Imports System.Drawing

Dim graphics As Graphics = Button1.CreateGraphics
Dim pen As Pen = New Pen(Color.Green)
graphics.DrawLine(pen, 0, 0, _
  Button1.Size.Width, Button1.Size.Height)
```

Drawing on a control this way, however, does not result in "permanent" drawings. When the control is repainted, as it is when the form containing the control is resized, the graphics will disappear. The section "The RoundButton Control" below explains how to make the user interface redraw every time the control is repainted.

Focus
Gives the focus to the control, making it the active control.

Hide
Set the control's `Visible` property to `False`, so that it is not shown.

GetNextControl
Returns the next control in the tab order.

On*Event*
Raises the *Event* event; possible events include `Click`, `ControlAdded`, `ControlRemoved`, `DoubleClick`,

`DragDrop`, `DragEnter`, `DragLeave`, `DragOver`, `Enter`, `GotFocus`, `KeyDown`, `KeyPress`, `KeyUp`, `LostFocus`, `MouseDown`, `MouseEnter`, `MouseHover`, `MouseLeave`, `MouseMove`, `MouseUp`, `Move`, `Paint`, `Resize`, and `TextChanged`. For example, calling the `OnClick` method of the control will trigger its `Click` event.

Show
Sets the control's `Visible` property to `True`, so that the control is shown.

The `UserControl` Class

The `UserControl` class provides an empty control that can be used to create other controls. It is an indirect child of the `Control` class. The object hierarchy of this control is as follows.

```
System.Object
  System.MarshalByRefObject
    System.ComponentModel.Component
      System.Windows.Forms.Control
        System.Windows.Forms.ScrollableControl
          System.Windows.Forms.ContainerControl
            System.Windows.Forms.UserControl
```

The `UserControl` class inherits all of the standard positioning and mnemonic-handling code from the `ContainerControl` class. This code is needed in a user control.

The RoundButton Control

With `Control` and `UserControl`, it is very easy to develop a custom Windows control. Your custom control class inherits the `UserControl` class and, because the `UserControl` class is also a descendent of the `Control` class, your custom control will also inherit all of the useful methods, properties, and events from the `Control` class. Event handling, for example, is automatically inherited in your custom control, thanks to the `Control` class.

How you draw the user interface is particularly important. Whatever shape your custom control has, be aware that the control is repainted occasionally. Therefore, the user interface must be redrawn whenever your custom control is repainted. Considering that the `Control` class's `OnPaint` method is called every time the control is repainted, you can ensure that your custom control has a permanent look by overriding this method with a new `OnPaint` method that draws your custom control's user interface.

The code in Example 1 presents a custom control called `RoundButton`, which is a button that is, um, round. Figure 1 shows the `RoundButton` custom control on a form. The code for the form is given in Example 2. Basically, all you need to do is override the `OnPaint` method. The system passes a `PaintEventArgs` object to this method, from which you can obtain the control's `System.Drawing.Graphics` object. You can then use its methods to draw the user interface.

Listing 1: The RoundButton Control

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

namespace MyNamespace {

  public class RoundButton : UserControl {

    public Color backgroundColor = Color.Blue;
    protected override void OnPaint(PaintEventArgs e) {
```

```
      Graphics graphics = e.Graphics;

      int penWidth = 4;
      Pen pen = new Pen(Color.Black, 4);

      int fontHeight = 10;
      Font font = new Font("Arial", fontHeight);

      SolidBrush brush = new SolidBrush(backgroundColor);
      graphics.FillEllipse(brush, 0, 0, Width, Height);
      SolidBrush textBrush = new SolidBrush(Color.Black);

      graphics.DrawEllipse(pen, (int) penWidth/2,
        (int) penWidth/2, Width - penWidth, Height - penWidth);

      graphics.DrawString(Text, font, textBrush, penWidth,
        Height / 2 - fontHeight);

    }
  }
}
```

The code in Listing 1 is a bit of a surprise, isn't it? It's too simple to be true. Your class has only one method: `OnPaint`. In a nutshell, this method passes a `PaintEventArgs` object, from which a `System.Drawing.Graphics` object can be obtained. This `Graphics` object represents the draw area of your custom control. Draw whatever you want on this `Graphics` object, and it will be displayed as the user interface of your custom control.

In Windows programming, you need a pen to draw a shape, and sometimes a brush. To write text, you will also need a font. The following code in the `OnPaint` method creates a `System.Drawing.Pen` object with a tip width of 4.

```
      int penWidth = 4;
      Pen pen = new Pen(Color.Black, 4);
```

It then creates a Arial Font object with a height of 10.

```
      int fontHeight = 10;
      Font font = new Font("Arial", fontHeight);
```

The RoundButton control is shown in Figure 1.

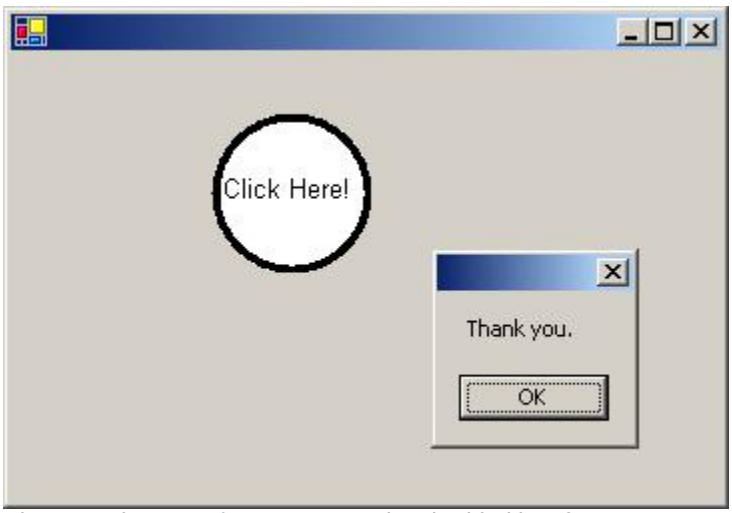

Figure 1: The `RoundButton` control embedded in a form.

The last bit of preparation is to instantiate a `SolidBrush` object having the same color as the value of the

`backgroundColor` field.

```
SolidBrush brush = new SolidBrush(backgroundColor);
```

Now you can start drawing. For the base, you use the `Graphics` class' `FillEllipse` method. The width and height of the circle are the same as the width and height of the control.

```
graphics.FillEllipse(brush, 0, 0, Width, Height);
```

Then, you instantiate another brush that you will use to draw text.

```
SolidBrush textBrush = new SolidBrush(Color.Black);
```

For the circle, you use the `DrawEllipse` method of the `Graphics` class.

```
graphics.DrawEllipse(pen, (int) penWidth/2,
  (int) penWidth/2, Width - penWidth, Height - penWidth);
```

Finally, you draw the text on the `Graphics` object using the `DrawString` method.

```
graphics.DrawString(Text, font, textBrush, penWidth,
  Height / 2 - fontHeight);
```

Compile your control into a *.dll* file and it's ready for use. The code in Example 2 presents a Windows form called `MyForm` that uses the `RoundButton` control.

Example 2: Using the `RoundButton` control

```
using System.Windows.Forms;
using System.Drawing;
using System;
using MyNamespace;


public class MyForm : Form {


  public MyForm() {
    RoundButton roundButton = new RoundButton();
    EventHandler handler = new EventHandler(roundButton_Click);
    roundButton.Click += handler;
    roundButton.Text = "Click Here!";
    roundButton.backgroundColor = System.Drawing.Color.White;
    roundButton.Size = new System.Drawing.Size(80, 80);
    roundButton.Location = new System.Drawing.Point(100, 30);
    this.Controls.Add(roundButton);


  }
  public void roundButton_Click(Object source, EventArgs e) {
    MessageBox.Show("Thank you.");
  }
  public static void Main() {
    MyForm form = new MyForm();
    Application.Run(form);
  }
}
```

The constructor instantiates a `RoundButton` object, creates an `EventHandler` object, and assigns the handler to the `Click` event of the `RoundButton` control.

```
    RoundButton roundButton = new RoundButton();
    EventHandler handler = new EventHandler(roundButton_Click);
    roundButton.Click += handler;
```

Note that we did not define any event in the `RoundButton` class. Event-handling capability is inherited from the `Control` class.

The next thing to do is to set some of the properties of the `RoundButton` control.

```
    roundButton.Text = "Click Here!";
    roundButton.backgroundColor = System.Drawing.Color.White;
    roundButton.Size = new System.Drawing.Size(80, 80);
    roundButton.Location = new System.Drawing.Point(100, 30);
```

And finally, add the control to the `Controls` collection of the form.

```
    this.Controls.Add(roundButton);
```

The `Click` event, when invoked by the user clicking the control, calls the `roundButton_Click` event handler, which simply displays a message box:

```
public void roundButton_Click(Object source, EventArgs e) {
    MessageBox.Show("Thank you.");
  }
```

Conclusion

In this article, you have been introduced to the two important classes in the `System.Windows.Forms` namespace that you should understand when building a custom control: `Control` and `UserControl`. You have also learned to build your own custom control by directly extending the `UserControl` class and how to use your custom control in a Windows form.

Budi Kurniawan is an IT consultant specializing in Internet and object-oriented programming, and has taught both Microsoft and Java technologies.

Return to the .NET DevCenter.