

Data Validation in the FCL

Consider again the example program we used when we first studied dialog boxes. This program has a modal dialog with two text boxes that allow you to specify the width and height of a rectangle.

You will see that in the property sheet of the edit boxes, there is a property *CausesValidation* that is set to *true*. Highlighting that property, you will see at the bottom of the property sheet the helpful information that this property “indicates whether the control causes and raises validation events”. Here is the documentation for the *Validating* and *Validated* events:

Focus events occur in the following order:

1. [Enter](#)
2. [GotFocus](#)
3. [Leave](#)
4. **Validating**
5. [Validated](#)
6. [LostFocus](#)

If the [CausesValidation](#) property is set to **false**, the **Validating** and **Validated** events are suppressed.

If the [Cancel](#) property of the [CancelEventArgs](#) object is set to **true** in the **Validating** event delegate, all events that would normally occur after the **Validating** event are suppressed.

This means that, in order to make sure the user can't enter a non-integer in the *WidthBox* textbox, we need to make that textbox handle the *Validating* event. Here's a first try at the code:

```
private void WidthBox_Validating(object sender,
System.ComponentModel.CancelEventArgs e)
{
    try{
        Convert.ToInt32(WidthBox.Text);
    }
    catch
    { e.Cancel = true;
      WidthBox.Select(0, WidthBox.Text.Length);
    }
}
```

Rather than try to examine the contents of the string ourselves, we just try to convert it to an integer, as we'll do anyway when the dialog terminates. If the conversion throws an exception, we set *e.Cancel* to true, which will prevent the focus from leaving that textbox! Run the program, type *cat* into that text box, and you'll see you can't tab or click in the other text box until you enter a valid integer. As an indication of the error, we select the text that needs correction.

Now, there are some problems with that:

- (1) it still lets you enter a negative integer, or an integer too large to be a sensible rectangle width.
- (2) After you have typed in "cat", you now can't even *Cancel* out of the dialog. You should always be able to *Cancel*!
- (3) When a string that does not convert to legal input has been entered, if the user doesn't realize what the problem is, that might be very frustrating; just selecting the text might not be enough of an error report.

We can fix (1) by adding more code in the try-block after the conversion succeeds; if, for example, the result is negative or too large, we can set *e.Cancel = true* in those cases also.

We can fix (2) by setting the *CausesValidation* property of the *Cancel* button to false. How does this work? The underlying Win32 message contains the information as to what control is *losing* the focus, and also what control is *gaining* the focus. The documentation quoted above is incomplete. The fact is,

FCL creates the *Validating* event only when focus is leaving the control being validated and going to a control with the *CausesValidation* property set to *True*.

If your dialog has a *Help* button, you should set its *CausesValidation* property to *False* also, so the user can get help about an invalid entry.

As for (3), there is a class in the FCL called *ErrorProvider* designed to solve this problem. Let's look at that now.

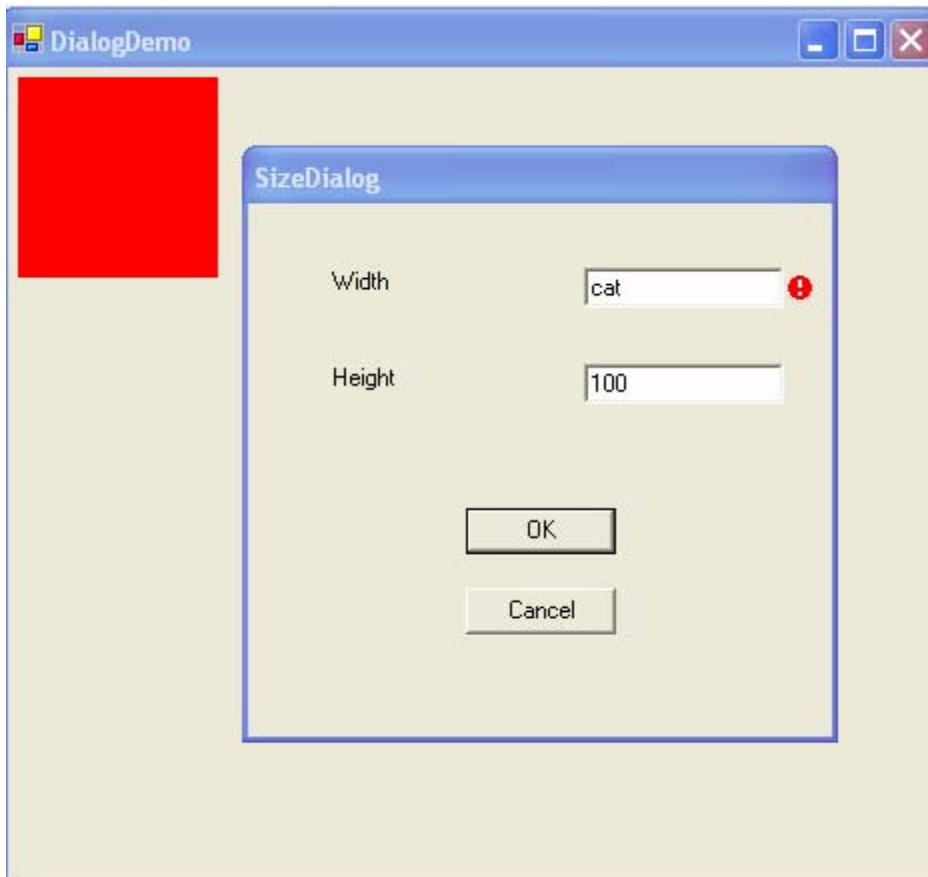
The *ErrorProvider* Class

From the Toolbox, you can drag and drop an *ErrorProvider* object onto your form. You'll need one for each field to be validated. Rename it (in its property sheet) to identify the control with which it is to be associated. For example, in our case, *WidthErrorProvider*. Here's how to use it:

```
private void WidthBox_Validating(object sender,
System.ComponentModel.CancelEventArgs e)
{
    try
    {
        Convert.ToInt32(WidthBox.Text);
    }
    catch
    {
        e.Cancel = true;
        WidthErrorProvider.SetError(WidthBox, "You must enter a
number");
    }
}
```

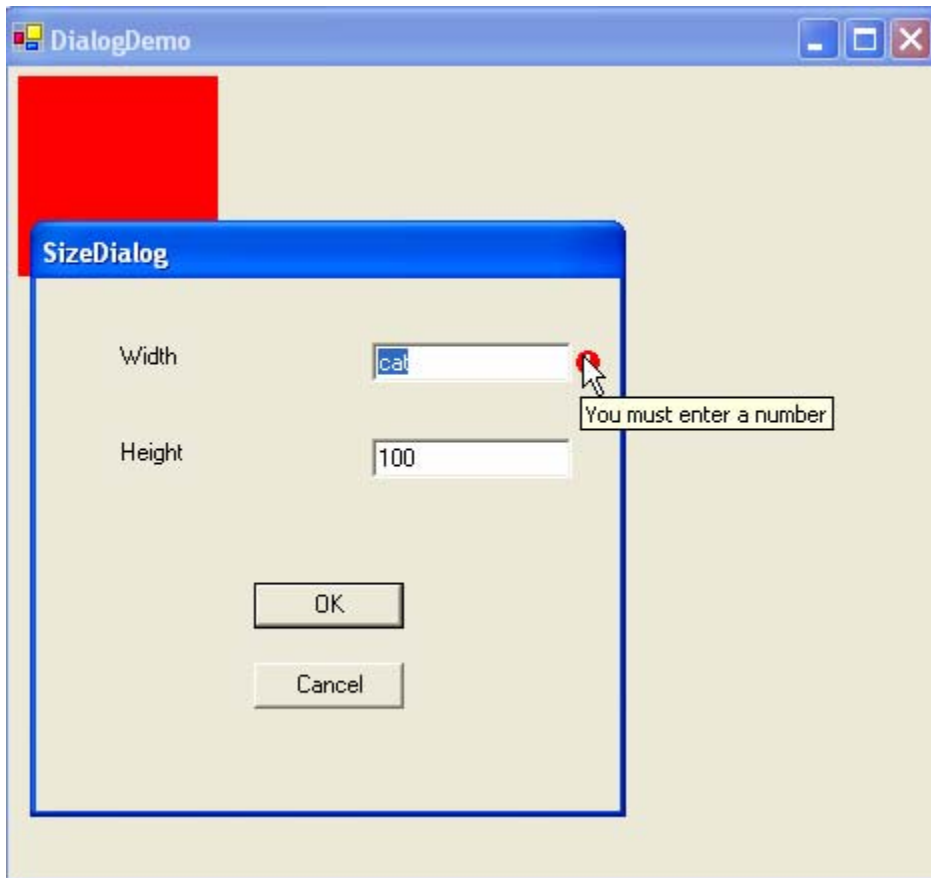
```
// MessageBox.Show("You must enter a number", "Error");  
}  
}
```

Try this code out: enter “cat” in the *WidthBox*, and press the tab key or click in the *Height* field. You’ll see this:



and the little red-and-white exclamation point will blink!

Now move the cursor over the blinking icon to see what the matter is:



A little tool-tip style window pops up with the error message. Now that is wonderful. But before you turn this program over to the sales department, try typing “100.73” into the *WidthBox*. That’s a number, isn’t it? But it doesn’t convert to an integer, so it still doesn’t validate, and the error message *You must enter a number* isn’t appropriate. It’s up to you, the programmer, to make sure the text of error messages corresponds to the validity check that failed.

This method of data validation may seem complex, but considering the alternatives, it is pretty nice. In the raw Windows API, data validation is quite challenging—there is no *Validating* message corresponding to this FCL event. MFC tried to make it much easier, but the limited forms of validation that it offered often created more trouble than they saved. The way FCL does it gives you complete control over the validation code itself, and makes it

fairly easy to have it called at the right time. In MFC, you don't have complete control, and in the raw API, you do have control but it's a lot of work. The *ErrorProvider* class is a really valuable tool that should be used in any professionally written dialog.

Validating several fields at once

The above example only shows how to validate one field at a time. Let's say that you wanted, for example, to insist that *Width* must be greater than *Height*. That can't be enforced on just one of the fields, because the user may be intending to change them both. This validation must be done when processing the *OK* button. That button also has the *CausesValidation* property set to true, but that is only so that *other* controls can get validated when the focus moves from them to the *OK* button. You can't do anything useful in the *Validating* handler of the *OK* button, because it's not called until the *OK* button's *Click* handler has already closed the dialog!

Therefore, you must do your overall data validation in the *OK* button's *Click* handler itself. Even if you don't have any multiple-field conditions to check (such as checking that *Width* is greater than *Height*), you may have fields that initially were blank, such as a field for an address, and if the user presses *OK* without ever once entering that field, it will not have been validated.

Therefore, the first thing we want to do in *OKButton_Click* is to validate each control individually (although in the case at hand, that is superfluous). Then, we add another *ErrorProvider* object and perform any multiple-field checking. Here's code to do this:

```

private void OKButton_Click(object sender,
                            System.EventArgs e)
{ // validate each control individually
  foreach(Control control in this.Controls)
  { control.Focus(); // set the focus
    if(!this.Validate())
      // validate the control that just lost focus
      { this.DialogResult = DialogResult.None;
        // prevent the dialog from closing
        return;
      }
  }
  int w = Convert.ToInt32(WidthBox.Text);
  int h = Convert.ToInt32(HeightBox.Text);
  // they must convert since they validated
  if (h > w)
  { OKErrorProvider.SetError( WidthBox,
    "height must be less than or equal to width.");
    this.DialogResult = DialogResult.None;
    return;
  }
}

```

This code only validates the controls that are directly children of the dialog. If we had a group box that contained some other controls, those controls would not be validated, since they would be children of the group box. Of course, if your group box contained only radio buttons, that wouldn't be a problem; only controls that contain text can contain invalid data. It is possible to write very general code that returns a list of all controls contained in children of children of children of the dialog, to any depth, but that would distract us from the basics of data validation, and anyway, in my opinion, if you have deeply nested controls, you probably need to redesign your form.

The *ErrorProvider* notification icon in the above code comes up, apparently, at the last control to lose the focus. I could not find a way to specify its location on the dialog; but that is relatively unimportant.