

## Modal Dialogs

Dialog boxes are windows containing some controls that are used for these purposes:

- collect data from user
- present data to the user
- permit user to control application data

Dialog boxes come in three flavors:

*Modal dialogs*: must be dismissed by user before any work can be done outside the dialog in the same application. (But you can work in another application).

*System modal*: can't even click to another application (example, the Control-Alt-Delete dialog).

*Modeless*: dialog can stay up while you work elsewhere (example, toolbars or tool palettes).

Some .NET Windows Forms applications have controls in the main window; such an application can be called “dialog-based”. Since the window in such an application is never “dismissed” (until the application is closed), this is not a modal dialog; but of course such an application can *also* make use of modal dialogs.

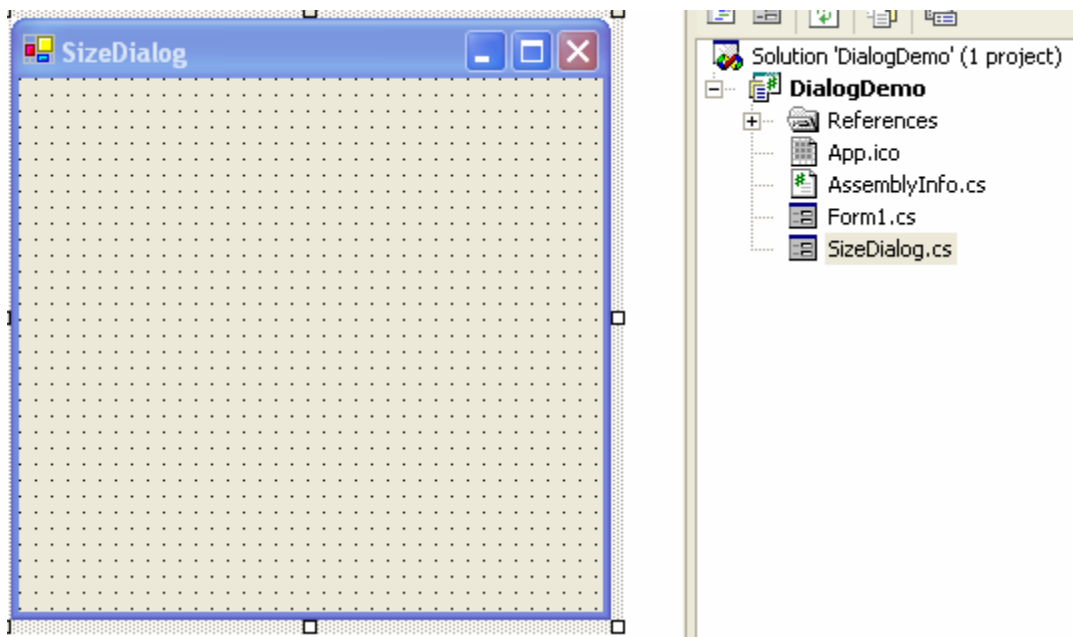
Modal dialogs are used with an ordinary SDI or MDI application. When you choose a “dialog-based” application, your main window is a dialog box that is never dismissed. This is not a modal dialog. Of course, such an application can *also* make use of modal dialogs, but the main window is not a modal dialog. You will see an example of this in the next lab exercise.

## Creating a modal dialog

Let's make the simplest possible dialog box, with just two controls, to let the user specify the height and width of a rectangle. Then in *Form1\_Paint*, we will draw a rectangle in the specified size.

Make a new project, *DialogDemo*. Add a member variable *m\_theRect* of type *Rectangle* and initialize it somehow, and add a *Paint* handler that calls *FillRectangle* to paint the rectangle red.

In .NET, the word “form” means “top-level window or dialog”, so what we want to do is add a new form. From the Visual Studio menu choose *Project / Add Windows Form / Windows Form*. Don't click OK yet! In the Name box, type *SizeDialog.cs*. Now click OK. This will be your modal dialog. You should see this:



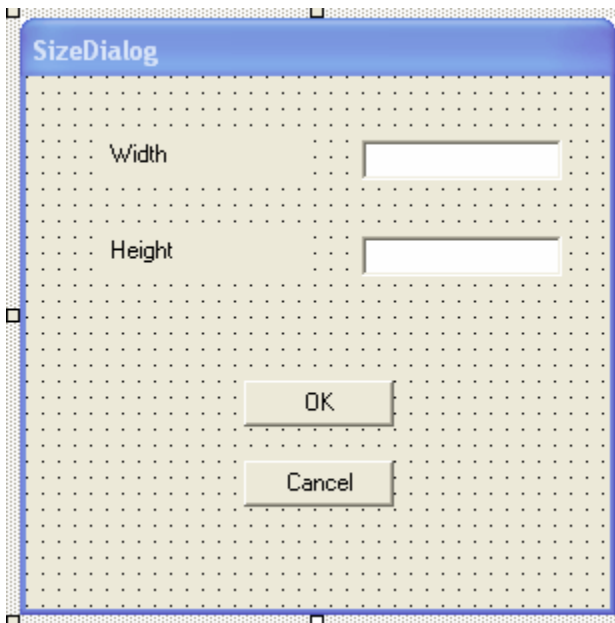
You see a new form in the form editor, and a corresponding C# source file *SizeDialog.cs* has been added to the project.

First, edit the properties of the form itself as follows:

*FormBorderStyle* should be *FixedDialog*  
*ControlBox*, *MaximizeBox*, *MinimizeBox*, and *ShowInTaskbar*  
should all be *false*.

This makes the dialog box be not resizable, not have a close box or maximize box or minimize box, and not show up as an icon in the taskbar (where we want only applications).

Next, design the dialog, by dragging and dropping two labels, two push buttons, and two text boxes from the Toolbox to the dialog. Edit the *Text* properties of the controls. Use the tools under *Format / Align* in Visual Studio's menu to help position the controls. We want it to look like this:



Here is how to set the properties of the controls:

For the labels: *Text* should be *Width*, *Height*, respectively.  
*Name* should be *WidthLabel*, *HeightLabel*.  
*TabIndex* should be 0,2 respectively.

For the text boxes: *Text* should be empty (blank).  
*Name* should be *WidthBox*, *HeightBox*.  
*TabIndex* should be 1,3, respectively.  
*Multiline* should be *False*.

For the OK button: *Text* should be *OK*  
*Name* should be *OKButton*  
*TabIndex* should be 4  
*DialogResult* should be *OK*

For the Cancel button: *Text* should be *Cancel*  
*Name* should be *NotOKButton*  
*DialogResult* should be *Cancel*  
*Text* should be *Cancel*.  
*TabIndex* should be 5

**WARNING:** Do not set the *Name* property of the *Cancel* button equal to *CancelButton*. That will conflict with a predefined object of that name (you will see that object later on in this lecture). In general, when we have *using* commands at the top of the file, strange errors can arise from accidentally naming some control with a name that conflicts with a predefined identifier that you don't know about. The best defense against this is to end your control names with endings like *Button* or *Box* as is done in the example.

Next we need to arrange that the text entered in the edit boxes is accessible from outside the dialog class. Here's the code that does that: (The darker shading indicates what you have to add; the lighter shaded part is included to show where to put it.)

```

public class SizeDialog : System.Windows.Forms.Form
{
    public String RectWidth
    {
        get { return WidthBox.Text; }
        set { WidthBox.Text = value; }
    }
    public String RectHeight
    {
        get { return HeightBox.Text; }
        set { HeightBox.Text = value; }
    }
}

```

Because these methods are declared *public*, you'll be able to access the *RectHeight* and *RectWidth* properties from outside this class. (I think you could simply make *HeightBox* and *WidthBox* public and then access their *Text* properties directly, but if you use the Visual Studio Form Editor, it makes them *private*.)

Now, we need to bring this dialog up somehow. The easiest thing for a demo is to bring the dialog up when the user right-clicks. To focus attention on the dialog code, we just accept any click, left or right, in this version.

Here's the code to do it:

```

private void Form1_MouseDown(object sender,
System.Windows.Forms.MouseEventArgs e)
{
    SizeDialog dlg = new SizeDialog();
    dlg.RectHeight = Convert.ToString(m_theRect.Height);
    dlg.RectWidth = Convert.ToString(m_theRect.Width);
    if (dlg.ShowDialog() == DialogResult.OK)
    { // get the data from the dialog
      // and convert Strings to int
      m_theRect.Width = Convert.ToInt32(dlg.RectWidth);
      m_theRect.Height= Convert.ToInt32(dlg.RectHeight);
      Invalidate();
    }
}
}

```

This is the crucial piece of code in the application, and the one you'll need to practice many times so that you can do it without hesitation. What does it do?

- Creates an object of the dialog class we defined.
- Initialize the members of that class using the member variables of the application.
- Calls the *ShowDialog* method of that class. This function returns only *AFTER* the user dismisses the dialog. That is, execution does not advance to the next line of code until then. Meantime, your dialog class is handling events.
- Checks the return value. That should be usually one of a few predefined values such as *DialogResult.OK* or *DialogResult.Cancel*.
- If the result is *DialogResult.OK* then extract the data from the dialog and store it in your application's member variables.
- If the changes necessitate redrawing, call *Invalidate*.

These are the basic steps required to program a modal dialog. You must practice them over and over, because creating modal dialogs is an extremely common requirement in Windows programming. You should be able to do this in your sleep.

I repeat for emphasis that *ShowDialog* does not return until the dialog is dismissed. Your *Form1* will not get any keyboard or mouse input during this time—it will all go to the dialog. However, your *Form1* can continue to get *Paint* events during this time.

In the example program, we chose to make the *RectWidth* property a string, and convert the result to an int back in *Form1*. We could have made it an int, and done the conversions in the get and set

functions for that property. Note, using Visual Studio's Intellisense, the wonderful functions in the *Convert* class.

Here are some common mistakes with the above code:

- Forgetting to initialize the dialog members before calling *ShowDialog*. This results in blank edit boxes, in this example.
- Forgetting to check the return value of *ShowDialog*. Then the dialog changes data even if *Cancel* is pressed.
- Not properly getting data from the dialog and storing it in application member variables. Then the dialog doesn't have the effect it is supposed to have.

## Keyboard Interface to Dialogs

The example so far still has several defects that we will fix. Here's the first: When you bring the dialog up, you have to click in one of the edit boxes before you can enter anything. It would be better if you could just start typing into the first edit box.

Here's another defect: the *Escape* key doesn't work to exit the dialog. In a good dialog, *Escape* should have the same effect as the *Cancel* button.

A third defect: Click in one of the text boxes, and then press the *Enter* key. That should have the same effect as pressing *OK*, but instead, it just makes an unpleasant noise.

The latter two defects can be fixed by this code:

```
AcceptButton = OKButton;  
CancelButton = NotOKButton;
```

Place this code in the dialog constructor, after *InitializeComponent*.

This has the effect of telling Windows that the *Escape* key should duplicate the *Cancel* button and (when the cursor is in an edit box) *Enter* should duplicate the *OK* button.

At any moment when Windows is running, some window “has the focus”, or more specifically, “has the keyboard focus”. That means that any keyboard input will go to that window. Remember that controls are windows, too. What *AcceptButton = OKButton* actually does, is to tell Windows that if some non-button control has the keyboard focus, then *Enter* will duplicate the *OK* button. But if a button has the keyboard focus, then *Enter* will just duplicate *that* button.

To fix the first defect, we want to set the focus to the first edit box in the dialog’s constructor. According to the documentation, this should be fixed by

```
WidthBox.Focus();
```

placed in the dialog constructor after *InitializeComponent*, but it didn’t work for me, and I haven’t solved that problem yet.

Finally, the tab key is supposed to take you through the controls in some sensible order. If you set the *TabIndex* properties as indicated above, this should be working. One way to avoid worrying about the *TabIndex* property is just to drag and drop the controls onto your dialog in the desired tab order. Then they will automatically be correctly numbered.

## **Data Validation**

Now try clicking OK with a blank in one of the edit boxes, or with “cat” and “dog” in the edit boxes. The program crashes! This is not very professional, to say the least. Also, it allows you to enter a negative number, but then of course no rectangle is drawn. It would be better to “validate the data” when the OK button is pressed, and not accept invalid data. There are a number of issues to consider about data validation, and a number of programming techniques and events in the FCL to assist you. This will be the subject of a separate lecture.