Chapter 6  
Dialogs

We all know the importance of dialogs in Windows applications. Dialogs using the .NET FCL are very easy to implement if you already know how to use basic controls on forms. A dialog is just a form that is configured with some specific property settings and is used in a slightly different way than you program’s main form.

Creating a Dialog Style Form

The first step is to create an empty form. This can most easily be done by right clicking the project name and then selecting Add|New Item. When the new item dialog appears select Windows Form and change the name to something appropriate, for example, “OptionsDialog.” This is shown in Figure 6-1. The new form will appear in the solution explorer. Now set the form’s properties according to Table 6-1. Also change the Text property to whatever is appropriate.

Figure 6-1

Table 6-1 Dialog Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FormBorderStyle</td>
<td>FormBorderStyle.FixedDialog</td>
</tr>
<tr>
<td>ControlBox</td>
<td>false</td>
</tr>
</tbody>
</table>
Figure 6-2 Shows what our empty dialog form will look like. Notice that there is no control menu button, minimize button, or a maximize button. The border style is also changed such that it does not allow resizing the form.

You can now place whatever controls you want on your dialog by dragging from the toolbox just as you do for any form. When you are finished you can resize the dialog to fit the controls. The usual layout tools apply. So far we really just have a standard form with some minor property changes. The real action takes place when we want to display the dialog.

**Displaying a Modal Dialog**

The standard method of displaying a dialog is to display it as a modal dialog. A modal dialog is a dialog that has to be closed in order for control to return to the main form. In other words you can’t click on the main form until the dialog is dismissed. We usually provide a pair of buttons, an OK button and a Cancel button. The intent is normally that if the Cancel button is clicked then all actions related to the dialog are ignored. If the OK button is clicked then the actions supported by the dialog are instituted. We now have two procedures to discuss. How do we initially display the dialog and how do we cause the OK and Cancel buttons to dismiss the dialog with the appropriate action or not.
A modal dialog is easily displayed using the `ShowDialog` method of the `Form` class. If our dialog is named “myDialog” then this simple statement does the trick:

```csharp
myDialog.ShowDialog();
```

Without an OK or Cancel button we can still close the dialog by typing Alt-F4. The proper way to close a dialog is to set the `DialogResult` property to any of the values shown in Table 6-2. The `ShowDialog` method returns this value so you can evaluate it to take the appropriate actions or ignore the dialog entirely.

### Table 6-2 DialogResult Enumeration

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abort</td>
<td>The dialog box return value is Abort (usually sent from a button labeled Abort).</td>
</tr>
<tr>
<td>Cancel</td>
<td>The dialog box return value is Cancel (usually sent from a button labeled Cancel).</td>
</tr>
<tr>
<td>Ignore</td>
<td>The dialog box return value is Ignore (usually sent from a button labeled Ignore).</td>
</tr>
<tr>
<td>No</td>
<td>The dialog box return value is No (usually sent from a button labeled No).</td>
</tr>
<tr>
<td>None</td>
<td>Nothing is returned from the dialog box. This means that the modal dialog continues running.</td>
</tr>
<tr>
<td>OK</td>
<td>The dialog box return value is OK (usually sent from a button labeled OK).</td>
</tr>
<tr>
<td>Retry</td>
<td>The dialog box return value is Retry (usually sent from a button labeled Retry).</td>
</tr>
<tr>
<td>Yes</td>
<td>The dialog box return value is Yes (usually sent from a button labeled Yes).</td>
</tr>
</tbody>
</table>

You might get a little confused here. Is `DialogResult` a property or an enumeration? Actually it’s both. It’s a property of the `Form` class and it’s an enumeration. The syntax in which we use it determines our intent. The compiler is able to figure this out as long as our syntax is correct.

Let’s try a simple example and carry it a little further. I will demonstrate how we can pass data to a dialog to initialize the controls and how we can extract data from the dialog after we close the dialog, I created a dialog with a single text box. Figure 6-3 shows the layout of this dialog. remember to set the properties specified in Table 6-1 Dialog Properties.
Figure 6-4 Shows the code for our dialog class. It is very sparse since most of the work has been done by setting the dialog and control properties. A click event handler has been added for the two buttons. They merely set the DialogResult property accordingly. The name of the text box is data. If you want to access this control outside the class itself you must be sure to modify the Modifiers property to public. This changes the declaration of the control to make it public. That’s about it for the dialog.

**Dialog1 – Dialog1.cs**

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace Dialog1
{
    public partial class Dialog : Form
    {
        public Dialog()
        {
            InitializeComponent();
        }

        private void OKbutton_Click(object sender, EventArgs e)
        {
            DialogResult = DialogResult.OK;
        }

        private void Cancelbutton_Click(object sender, EventArgs e)
        {
            DialogResult = DialogResult.Cancel;
        }
    }
}
```
The main form contains only a single menu strip with an appropriate menu item to display the menu. This is shown in Figure 6-5. Figure 6-6 show the code. The `click` event for the Display menu item calls the `ShowDialog` method. A `private` string, `boxtext`, is used to retain the value of a string entered on the form. You already know how to use basic controls. The only difference here is that the controls are not in our main form, but rather in our dialog. If the dialog is created every time we need it, then we must make sure to save the dialog data someplace. Usually dialog data is linked to the overall state and actions of your application so what you need to do should unfold quite easily.

In this case we take the value of `boxtext` and store it in the text box of our dialog. This can’t be done until we instantiate the `Dialog` class object. After `ShowDialog` returns the return value is checked to see if it corresponds to the OK button. If it does, then we save the text box value back into `boxtext` for use later.

```csharp
namespace Dialog1
{
    public partial class Form1 : Form
    {
    }
```
private string boxtext = "";

public Form1()
{
    InitializeComponent();
}

private void dIsplayToolStripMenuItem_Click(object sender, EventArgs e)
{
    Dialog myDialog = new Dialog();
    myDialog.data.Text = boxtext;
    if (myDialog.ShowDialog() == DialogResult.OK)
        boxtext = myDialog.data.Text;
}

Figure 6-6

**DialogResult Shortcut**

There is an alternative to providing *click* event handlers for the OK, Cancel, or other buttons that close the dialog. We can't use this shortcut if we want to do any processing when the button is clicked other than set the DialogResult property. However, quite often we have no need to perform any steps. This is especially true of the Cancel button.

All we need to do is to set the DialogResult property for each of the buttons to the appropriate value. This is shown in Figure 6-7. As you can see from Figure 6-8 our dialog contains no code other than that generated by the designer. Wow! We accomplished this all with our mouse and skillful use of the designer.

![Figure 6-7](image-url)
In the example we just discussed we made the text box control public so we could easily access the data. As we have already discussed it is not always desirable to allow public access to class members. C# properties provide us a real measure of security since we can assure that members are accessed with whatever restrictions we desire to impose. In Chapter <ref> I discussed how to convert the string value from a text box into an integer. This is a very common requirement of dialogs since we often provide integer values to and from text boxes. Whether or not you provide a property that corresponds to a control in your dialog is your decision.

Another design decision that often confronts the programmer is whether or not to create a dialog object only once for the life of the program, or to instantiate a new one each time we invoke the dialog. There are pros and cons for each technique. If a dialog is used very often we can save time by creating the dialog when the application starts. This can be done in the constructor for your main form class. On the other hand if we have many dialogs that might never be invoked, we can save startup time and conserve resources by creating the dialog only when needed. The designer has to weigh the tradeoffs. The time to create a dialog with many controls may be significant as the controls themselves encapsulate operating system resources. Figure 6-9 shows how we can create our dialog only once for the life of the program. There is no difference instantiating the object with an initializer or doing it in the constructor. This is one of the nice features of C# that is not found in C++.
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace Dialog3
{
    public partial class Form1 : Form
    {
        private string boxtext = "";
        private Dialog myDialog = new Dialog();

        public Form1()
        {
            InitializeComponent();
        }

        private void dIsplayToolStripMenuItem_Click(object sender, EventArgs e)
        {
            myDialog.data.Text = boxtext;
            if (myDialog.ShowDialog() == DialogResult.OK)
            {
                boxtext = myDialog.data.Text;
            }
        }
    }
}

Figure 6-9

Accept and Cancel

In Chapter <ref> I discussed the use of the TabIndex and TabStop properties of controls. If a button control has the focus then pressing the return key or typing a space is identical to clicking on it with your mouse. However, if a non-button control has the focus then hitting return has a different effect. You may get a beep sound if it is a text box that does not accept returns, or no effect at all for a check box.

It is very simple to associate the default action for the return key, if it is not processed by the control, to be directed to a specific button. In a similar manner we can associate the escape key with another button. The AcceptButton and CancelButton properties of the form can be set to the buttons of your choice. Figure 6-10 shows how you can set these properties in the properties window. Be sure to click on the form itself in the designer and not on a control. The last example above implements this feature so you can try it out.
The default position of a form or dialog is determined by the Windows operating system. You will probably soon determine that it is somewhat unpredictable. Windows tries not to have new windows exactly overlap existing windows. So what if you want to have exact control? Well you can. The `StartPosition` property of any form can be set to one of the values in Table 6-3 `FormStartPosition` Enumeration. `CenterParent` and `CenterScreen` display the form in the size you set with the designer. The form is centered in the parent form or the total screen respectively. `Manual` allows you to set the `Location` property of the form for an exact positioning. This is sometimes used to position the dialog in a specific position within the main form's bounds. For example, the upper left hand corner. The location is relative to the parent form and not the screen. `WindowsDefaultBounds` and `WindowsDefaultLocation` differ only in that the former sets the size of the form and the latter, the default, doesn’t. We rarely would want to use `WindowsDefaultBounds`.
Table 6-3 FormStartPosition Enumeration

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CenterParent</td>
<td>The form is centered within the bounds of its parent form.</td>
</tr>
<tr>
<td>CenterScreen</td>
<td>The form is centered on the current display, and has the dimensions specified in the form's size.</td>
</tr>
<tr>
<td>Manual</td>
<td>The position of the form is determined by the Location property.</td>
</tr>
<tr>
<td>WindowsDefaultBounds</td>
<td>The form is positioned at the Windows default location and has the bounds determined by Windows default.</td>
</tr>
<tr>
<td>WindowsDefaultLocation</td>
<td>The form is positioned at the Windows default location and has the dimensions specified in the form's size.</td>
</tr>
</tbody>
</table>

One question that needs to be addressed is exactly what is the parent container? If you create your own form then there is initially no parent container. This means that unless you indicate a parent container the CenterParent will not work as you expect and Manual positions will be relative to the screen and not your main form. However, if you show a modal dialog then the parent automatically becomes the main form and so CenterParent works as shown in Figure 6-11. Unfortunately this doesn’t work for modeless dialogs. I will discuss modeless dialog below.

Handling Events in a Dialog

Since a dialog is a form it can have handlers for events associated with its controls. Some buttons you frequently see in dialogs are Clear, Apply, and Defaults. In fact, a button in a dialog can invoke another dialog. The example
shown in Figure 6-12 through Figure 6-16 demonstrates such a dialog that changes the background color of the main dialog from our last example. Notice that I have added the necessary code and set the necessary properties to make sure that each dialog is centered in its parent.

BackColor is the form property that can be set to change the forms default background color. In the color dialog rather than make the radio buttons public I added a property that returns a Color object. This value can be stored directly in BackColor. Figure 6-16 shows the result of changing the background color to white.

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace Dialog4
{
    public partial class Dialog : Form
    {
        public Dialog()
        {
            InitializeComponent();
        }

        private void bColor_Click(object sender, EventArgs e)
        {
            ColorDlg cdlg = new ColorDlg();
            cdlg.Owner = this;
            cdlg.ShowDialog();
            BackColor = cdlg.color;
        }
    }
}
```

Figure 6-12

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace Dialog4
{
    public partial class ColorDlg : Form
    {
        public ColorDlg()
        {
            InitializeComponent();
        }

        private void bColor_Click(object sender, EventArgs e)
        {
            Color color = Color.FromArgb(255, 255, 255);
            BackColor = color;
        }
    }
}
```
```csharp
public partial class ColorDlg : Form
{
    public ColorDlg()
    {
        InitializeComponent();
    }
    public Color color
    {
        get
        {
            if (grayButton.Checked)
                return Color.LightGray;
            else return Color.White;
        }
    }
}
```
Modeless Dialogs

Unlike a modal dialog, a modeless dialog allows you to return to the main form and perform operations at the same time as the modeless dialog stays on the screen. Instead of invoking the modeless dialog with the `ShowDialog` method we use the `Show` method. One important thing to remember is that you can’t instantiate the dialog with a reference on the stack or you will be unable to access the dialog after you return from the method used to activate the dialog. The dialog will continue to exist as long as it is displayed because it has been created on the heap.

If you create the dialog by setting the properties as specified in Table 6-1 the only way to close the dialog is by typing Alt-F4 unless we provide a button similar to OK or Cancel. These buttons are usually not appropriate for modeless
dialogs and the `DialogResult` property has no effect. The proper way to provide a button, for example a Close button, is to invoke the `Hide` method. You might be tempted to use the `Close` method instead. That would be wrong since the `Close` method results in the destruction of the dialog object. We really want to just hide the dialog. Figure 6-17 Shows the code for a dialog that will change the size of the main form. It has a Resize button and a Close button. The event handler for the Resize button set the width and height of the main form using the `Owner` property which we must make sure to set prior to showing the form. If illegal values are entered in the form’s text boxes then a message box is displayed.

**ModelessDialog – SetSize.cs**

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace ModelessDialog
{
    public partial class SetSize : Form
    {
        public SetSize()
        {
            InitializeComponent();
        }

        private void close_Click(object sender, EventArgs e)
        {
            Hide();
        }

        private void resizeButton_Click(object sender, EventArgs e)
        {
            int w, h;
            try
            {
                w = Convert.ToInt32(width.Text);
                h = Convert.ToInt32(height.Text);
            }
            catch
            {
                MessageBox.Show("Invalad size!");
                return;
            }
            Owner.Width = w;
            Owner.Height = h;
        }
    }
}
```

*Figure 6-17*
Figure 6-18 shows our main form’s code. You can see that the `Show` method is used instead of the `ShowDialog` method. I have also demonstrated a shortcut way to set up the owner property of the dialog. Instead of setting this property directly we can merely pass the `this` reference to `Show`. This also works for the `ShowDialog` method. There is one other reason to set this property. If we don’t set the `Owner` property then the modeless dialog will not remain on top of the main form. We usually want this behavior. To invoke the dialog a right mouse button click is used by providing an event handler for the `MouseDown` event.

Figure 6-19 shows the dialog with the default 300 by 300 pixel window size. Figure 6-20 show the result of resizing to 400 by 400 pixels. There is a text box on the main form that helps demonstrate that you can return to the main form while the dialog is displayed and enter data. The modeless dialog can be moved anywhere on the screen and doesn’t have to overlap the main form. Try it.

One common requirement of a modeless dialog is to inform the main form that the user has changed something in the dialog. It is usually not the job of the modeless dialog to manipulate the application. It is a better strategy to inform the main form that the dialog has changed and that appropriate code should be executed to act on these changes. The solution is to trigger an event that is handled by the code in the main form. This is very much in the spirit of object oriented programming. Consider a find and replace dialog. The dialog should

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace ModelessDialog
{
    public partial class Form1 : Form
    {
        SetSize sizeDialog = new SetSize();
        public Form1()
        {
            InitializeComponent();
        }

        private void Form1_MouseDown(object sender, MouseEventArgs e)
        {
            if (e.Button == MouseButtons.Right)
            {
                sizeDialog.Show(this);
            }
        }
    }
}
```
provide the string to find and the replacement string, but the actual job of searching and replacing should be done by the user of the dialog and not the dialog. We normally use dialogs to provide information only. I will discuss the techniques to provide a custom event handler in Chapter <ref>.

Figure 6-19

Figure 6-20
Common Dialog Classes

Some dialogs are so common that they are actually included in the Windows operating system. The .NET FCL wraps these dialogs in a set of classes. There is a big advantage to using the dialogs in addition to the lack of work on your part. Most users are familiar with a particular look and feel for such common operations as opening files and selecting colors. If you were to design a dialog that is significantly different from the accepted standards then the user will either be confused or have a learning curve.

The abstract class `System.Windows.Forms.CommonDialog` is the base class for all the common dialogs. Table 6-4 lists all the classes derived from `System.Windows.Forms.CommonDialog`. `System.Windows.Forms.FileDialog` is actually the base class for the `OpenFileDialog` and `SaveFileDialog` classes.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>System.Windows.Forms.ColorDialog</code></td>
<td>Represents a common dialog box that displays available colors along with controls that allow the user to define custom colors.</td>
</tr>
<tr>
<td><code>System.Windows.Forms.FileDialog</code></td>
<td>Displays a dialog box from which the user can select a file.</td>
</tr>
<tr>
<td><code>System.Windows.Forms.FolderBrowserDialog</code></td>
<td>Prompts the user to select a folder. This class cannot be inherited.</td>
</tr>
<tr>
<td><code>System.Windows.Forms.FontDialog</code></td>
<td>Prompts the user to choose a font from among those installed on the local computer.</td>
</tr>
<tr>
<td><code>System.Windows.Forms.PrintDialog</code></td>
<td>Allows users to select a printer and choose which portions of the document to print.</td>
</tr>
</tbody>
</table>

I will discuss the file related dialogs in Chapter <ref> when I cover file input/output. The page setup dialog and the print dialog are best left for Chapter <ref> when printing is covered. The font and color dialogs are very simple to use.

The Font Dialog

Figure 6-21 is a simple program to demonstrate the font dialog. It displays a single line of text in the client area. If you right click anywhere inside the form the font dialog is displayed. You can then select the font you wasn’t the text displayed in. The `ShowDialog` method is used to display the font dialog just like
for a dialog you create yourself. You still need to instantiate the `FontDialog` class before you show it. The field `myFont` is used to save the current font. The form's constructor initializes this reference to the form's default font. In order to have the font dialog display the current font selection you must initialize the `Font` property in the dialog. This is a courtesy to the user who might want to just change the size of the font and not other characteristics.

Figures Figure 6-22 and Figure 6-23 show the font dialog and the result of setting the font to `Stencil` in an 18 point size. What is very convenient is that the font dialog creates the new font for you. This couldn't be easier.
The color dialog is just as easy to use as the font dialog. Figure 6-24 shows a simple use of the color dialog to change the color of the form's background. Unlike the previous example there is no need to save the color in a field since the color is saved in the BackColor property of the form. If we used the same technique in the previous example we would be changing the font for the entire form and not just for the text we were displaying. However, you should be aware that the BackColor property is inherited by any controls that you display in the client area of the form.

CommDlg2 - Form1.cs
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace CommDlg2
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        protected override void OnPaint(PaintEventArgs e)
        {
            Graphics g = e.Graphics;
            g.DrawString("Right click to change the form's background color.",
                        Font, Brushes.Black, 10, 10);
        }
        private void Form1_MouseClick(object sender, MouseEventArgs e)
        {
            ColorDialog colorDialog = new ColorDialog();
            colorDialog.Color = BackColor;
            if (e.Button == MouseButtons.Right)
            {
                if (colorDialog.ShowDialog() == DialogResult.OK)
                {
                    BackColor = colorDialog.Color;
                    Invalidate();
                }
            }
        }
    }
}

Figure 6-24

Figure 6-25 shows the color dialog. If you click on the Define Custom Colors button an expanded panel appears that allows you to create any arbitrary color. This is shown in Figure 6-26. Several methods can be used including selecting the red, green, and blue components of the color. Internally the Windows operating system represents colors in an eight bit value for each of these three color components. The vvalue of each component can range from 0 and 255.
Figure 6-25

Figure 6-26