

Chapter 13. Building Windows Applications

The previous chapters have used console applications to demonstrate C# and the Common Language Runtime. Although console applications can be implemented simply, it is time to turn your attention to the reason you're learning the C# language in the first place: building Windows and web applications.

In the early days of Windows computing, an application ran on a desktop, in splendid isolation. Over time, developers found it beneficial to spread their applications across a network, with the user interface on one computer and a database on another. This division of responsibilities or partitioning of an application came to be called two-tier or client-server application development. Later three-tier or *n*-tier approaches emerged as developers began to use web servers to host business objects that could handle the database access on behalf of clients.

When the Web first came along, there was a clear distinction between Windows applications and web applications. Windows applications ran on the desktop or a local area network (LAN), and web applications ran on a distant server and were accessed by a browser. This distinction is now being blurred as Windows applications reach out to the Web for services. Many new applications consist of logic running on a client, a database server, and remote third-party computers located on the Web. Traditional desktop applications such as Excel or Outlook are now able to integrate data retrieved through web connections seamlessly, and web applications can distribute some of their processing to client-side components.

The primary remaining distinction between a Windows application and a web application might be this: who owns the user interface? Will your application use a browser to display its user interface, or will the UI be built into the executable running on the desktop?

There are enormous advantages to web applications, starting with the obvious: they can be accessed from any browser that can connect to the server. In addition, updates can be made at the server, without the need to distribute new dynamic link libraries (DLLs) to your customers.

On the other hand, if your application derives no benefit from being on the Web, you might find that you can achieve greater control over the look and feel of your application or that you can achieve better performance by building a desktop application.

.NET offers closely related, but distinguishable, suites of tools for building Windows or web applications. Both are based on forms, with the premise that many applications have user interfaces centered on interacting with the user through forms and controls, such as buttons, list boxes, text, and so forth.

The tools for creating web applications are called Web Forms and are considered in [Chapter 15](#). The tools for creating Windows applications are called Windows Forms and are the subject of this chapter.



It is my prediction that the distinction between Web Forms and Windows Forms is temporary. There is such obvious similarity between these two approaches that I'd be very surprised if the next version of .NET didn't merge these two tools into one unified development environment.

In the following pages, you will learn how to create a simple Windows Form using either a text editor such as Notepad or the Design tool in Visual Studio .NET. Next you will build a more complex Windows application using Visual Studio, the Windows Forms framework, and a number of C# programming techniques you learned in earlier chapters. The chapter concludes with a brief introduction to Documentation Comments, a new XML-facilitated means to document applications, and an introduction to the deployment of .NET applications.

13.1 Creating a Simple Windows Form

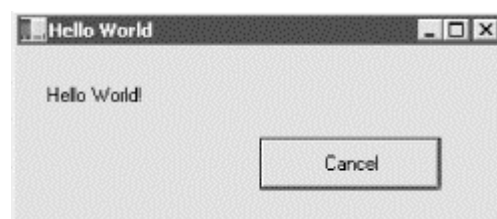
A Windows Form is a tool for building a Windows application. The .NET Framework offers extensive support for Windows application development, the centerpiece of which is the Windows Forms framework. Not surprisingly, Windows Forms use the metaphor of a form. This idea was borrowed from the wildly successful Visual Basic (VB) environment and supports Rapid Application Development (RAD). Arguably, C# is the first development environment to marry the RAD tools of Visual Basic with the object-oriented and high-performance characteristics of a C-family language.

13.1.1 Using Notepad

Visual Studio .NET provides a rich set of drag-and-drop tools for working with Windows Forms. It *is* possible to build a Windows application without using the Visual Studio Integrated Development Environment (IDE), but it is far more painful and takes a lot longer.

However, just to prove the point, you'll use Notepad to create a simple Windows Form application that displays text in a window and implements a Cancel button. The application display is shown in [Figure 13-1](#).

Figure 13-1. The hand-drawn Windows Form



You start by adding a `using` statement for the Windows Forms namespace:

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

The key to creating a Windows Form application is to derive your form from `System.Windows.Forms.Form`:

```
public class HandDrawnClass : Form
```

The `Form` object represents any window displayed in your application. You can use the `Form` class to create standard windows, as well as floating windows, tools, dialog boxes, and so forth. Microsoft apparently chose to call this a form rather than a window to emphasize that most windows now have an interactive component that includes controls for interacting with users.

All the Windows widgets you'll need (labels, buttons, list boxes, etc.) are found within the `Windows.Forms` namespace. In the IDE, you'll be able to drag and drop these objects onto a designer, but for now you'll declare them right in your program code.

To get started, declare the two widgets you need, a label to hold the `Hello World` text, and a button to exit the application:

```
private System.Windows.Forms.Label lblOutput;
private System.Windows.Forms.Button btnCancel;
```

You're now ready to instantiate these objects, which is done in the `Form`'s constructor:

```
this.lblOutput = new System.Windows.Forms.Label( );
this.btnCancel = new System.Windows.Forms.Button( );
```

Next you can set the `Form`'s title text to `Hello World`:

```
this.Text = "Hello World";
```



Note that the preceding statements appear in your form's constructor, `HandDrawnClass`, and so the `this` keyword refers to the form itself.

Set the label's location, text, and size:

```
lblOutput.Location = new System.Drawing.Point (16, 24);
lblOutput.Text = "Hello World!";
lblOutput.Size = new System.Drawing.Size (216, 24);
```

The location is expressed as a `System.Drawing.Point` object, whose constructor takes a horizontal and vertical position. The size is set with a `Size` object, whose constructor takes a pair of integers that represent the width and height of the object.



The .NET Framework provides the `System.Drawing` namespace, which encapsulates the Win32 GDI+ graphics functions. Much of the .NET Framework Class Library (FCL) consists of classes that encapsulate Win32 methods as objects.

Next, do the same for the `button` object, setting its location, size, and text:

```
btnCancel.Location = new System.Drawing.Point (150,200);
btnCancel.Size = new System.Drawing.Size (112, 32);
btnCancel.Text = "&Cancel";
```

The button also needs an event handler. As described in [Chapter 12](#), events (in this case the cancel button-click event) are implemented using delegates. The publishing class (`Button`) defines a delegate (`System.EventHandler`) that the subscribing class (your form) must implement.

The delegated method can have any name but must return `void` and take two parameters: an object (`sender`) and a `System.EventArgs` object, typically named `e`:

```
protected void btnCancel_Click (
    object sender, System.EventArgs e)
{
    //...
}
```

Register your event-handler method in two steps. First, create a new `System.EventHandler` delegate, passing in the name of your method as a parameter:

```
new System.EventHandler (this.btnCancel_Click);
```

Then add that delegate to the button's click event-handler list with the `+=` operator.

The following line combines these steps into one:

```
btnCancel.Click +=
    new System.EventHandler (this.btnCancel_Click);
```

Now you must set up the form's dimensions. The form property `AutoScaleBaseSize` sets the base size used at display time to compute the scaling factor for the form. The `ClientSize` property sets the size of the form's client area, which is the size of the form excluding borders and titlebar. (When you use the designer, these values are provided for you interactively.):

```
this.AutoScaleBaseSize = new System.Drawing.Size (5, 13);
this.ClientSize = new System.Drawing.Size (300, 300);
```

Finally, remember to add the widgets to the form:

```
this.Controls.Add (this.btnCancel);
this.Controls.Add (this.lblOutput);
```

Having registered the event handler, you must supply the implementation. For this example, clicking Cancel will exit the application, using the static method `Exit( )` of the `Application` class:

```
protected void btnCancel_Click (
    object sender, System.EventArgs e)
{
    Application.Exit ( );
}
```

That's it; you just need an entry point to invoke the constructor on the form:

```
public static void Main( )
{
    Application.Run(new HandDrawnClass( ));
}
```

The complete source is shown in [Example 13-1](#). When you run this application, the window is opened and the text is displayed. Pressing Cancel closes the application.

Example 13-1. Creating a hand-drawn Windows Form

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

namespace ProgCSharp
{
    public class HandDrawnClass : Form
    {
        // a label to display Hello World
        private System.Windows.Forms.Label
            lblOutput;

        // a cancel button
        private System.Windows.Forms.Button
            btnCancel;

        public HandDrawnClass( )
        {
            // create the objects
            this.lblOutput =
                new System.Windows.Forms.Label ( );
            this.btnCancel =
                new System.Windows.Forms.Button ( );

            // set the form's title
            this.Text = "Hello World";

            // set up the output label
            lblOutput.Location =
                new System.Drawing.Point (16, 24);
            lblOutput.Text = "Hello World!";
            lblOutput.Size =
                new System.Drawing.Size (216, 24);

            // set up the cancel button
            btnCancel.Location =
                new System.Drawing.Point (150,200);
            btnCancel.Size =
                new System.Drawing.Size (112, 32);
            btnCancel.Text = "&Cancel";

            // set up the event handler
            btnCancel.Click +=
                new System.EventHandler (this.btnCancel_Click);

            // Add the controls and set the client area
            this.AutoScaleBaseSize =
                new System.Drawing.Size (5, 13);
            this.ClientSize =
                new System.Drawing.Size (300, 300);
            this.Controls.Add (this.btnCancel);
        }
    }
}
```

```

        this.Controls.Add (this.lblOutput);
    }

    // handle the cancel event
    protected void btnCancel_Click (
        object sender, System.EventArgs e)
    {
        Application.Exit( );
    }

    // Run the app
    public static void Main( )
    {
        Application.Run(new HandDrawnClass( ));
    }
}

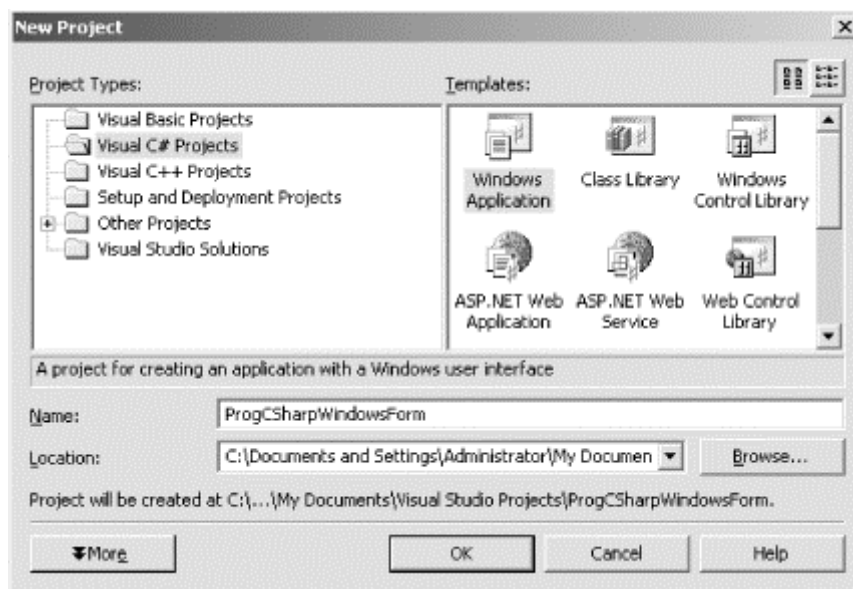
```

13.1.2 Using the Visual Studio .Net Designer

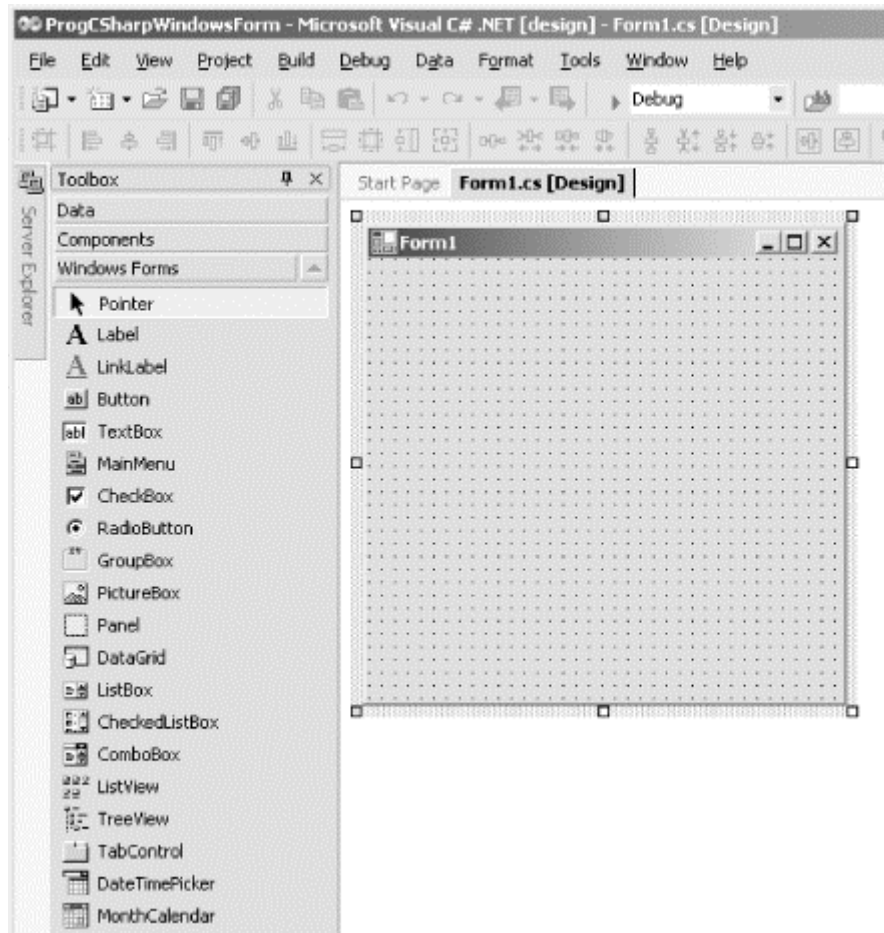
Although hand coding is always great fun, it is also a lot of work, and the result in the previous example is not as elegant as most programmers would expect. The Visual Studio IDE provides a design tool for Windows Forms that is much easier to use.

To begin work on a new Windows application, first open Visual Studio and choose New Project. In the New Project window, create a new C# Windows application and name it ProgCSharpWindowsForm, as shown in [Figure 13-2](#).

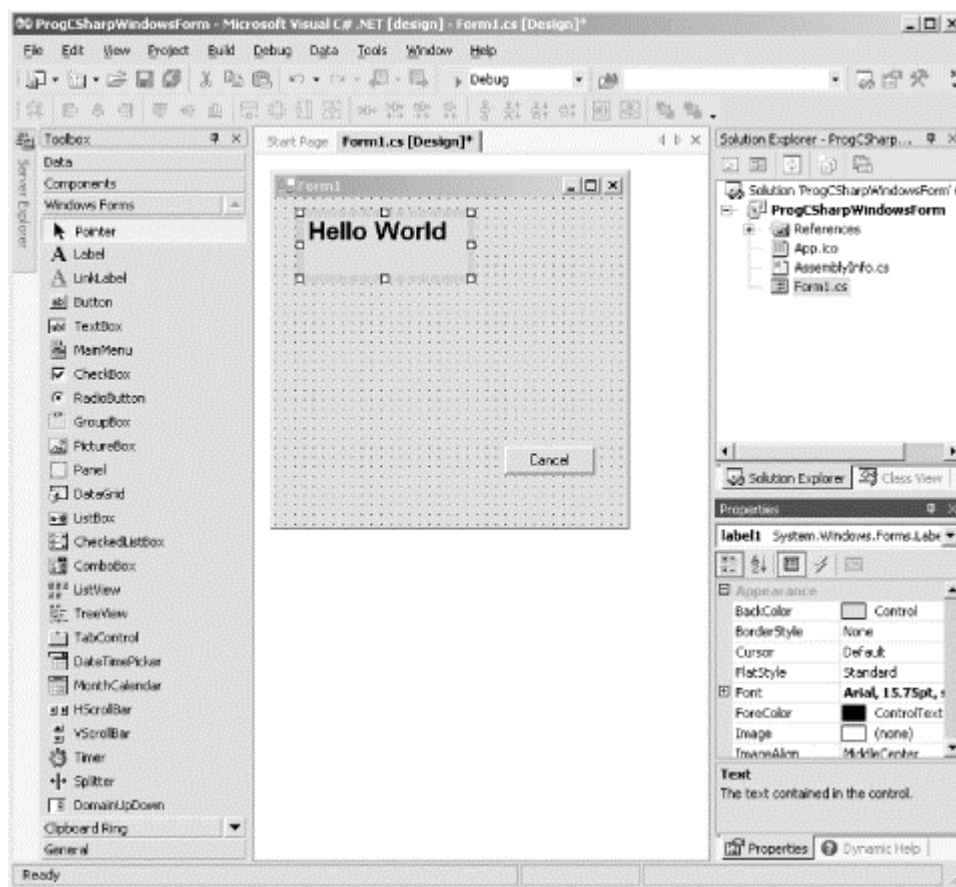
Figure 13-2. Creating a Windows Form application



Visual Studio responds by creating a Windows Form application, and, best of all, putting you into a design environment, as shown in [Figure 13-3](#).

Figure 13-3. The design environment

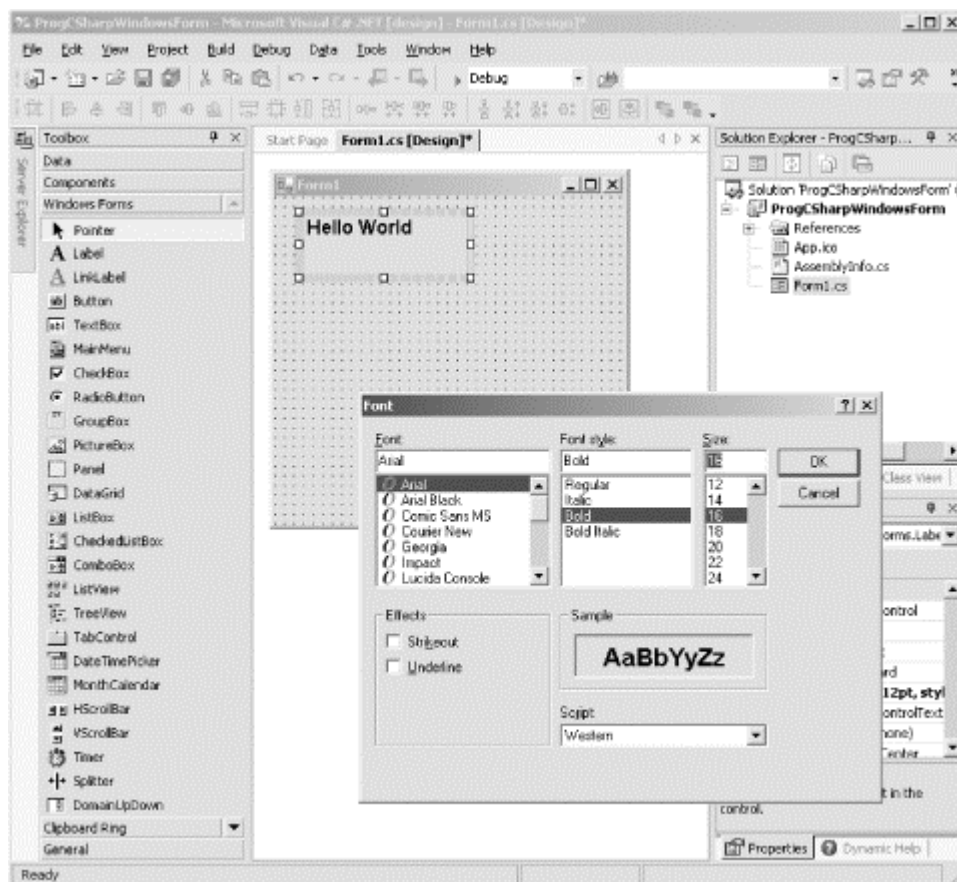
The Design window displays a blank Windows Form (`Form1`). A Toolbox window is also available, with a selection of Windows widgets and controls. If the Toolbox is not displayed, try clicking the word "Toolbox," or select **View** → **Toolbox** on the Visual Studio menu. You can also use the keyboard shortcut **Ctrl-Alt-X** to display the Toolbox. With the Toolbox displayed, you can drag a label and a button directly onto the form, as shown in [Figure 13-4](#).

Figure 13-4. The Windows Form development environment

Before proceeding, take a look around. The Toolbox is filled with controls that you can add to your Windows Form application. In the upper-right corner you should see the Solution Explorer, which is a window that displays all the files in your projects. In the lower-right corner is the Properties window, which displays all the properties of the currently selected item. In [Figure 13-4](#), the label (`label1`) is selected, and the Properties window displays *its* properties.

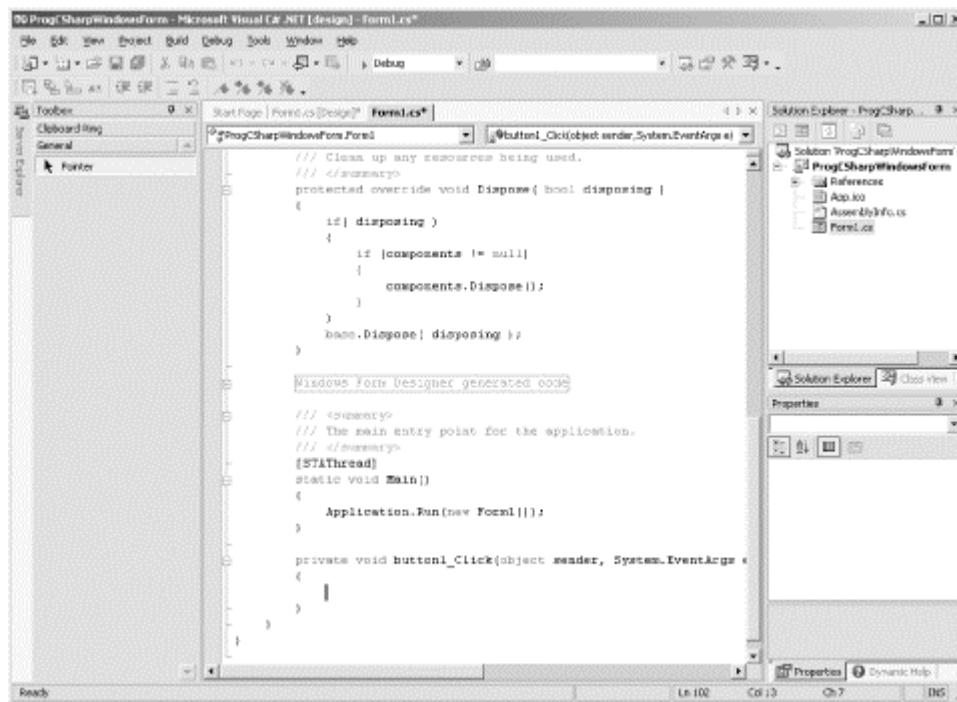
You can use the Properties window to set the static properties of the various controls. For example, to add text to `label1`, you can type the words "Hello World" into the box to the right of its `Text` property. If you want to change the font for the lettering in the `HelloWorld` label, click the `Font` property shown in the lower-right corner of [Figure 13-5](#). (You can provide text in the same way for your button (`button1`) by selecting it in the Property window and typing the word "Cancel" into its `Text` property.)

Figure 13-5. Modifying the font



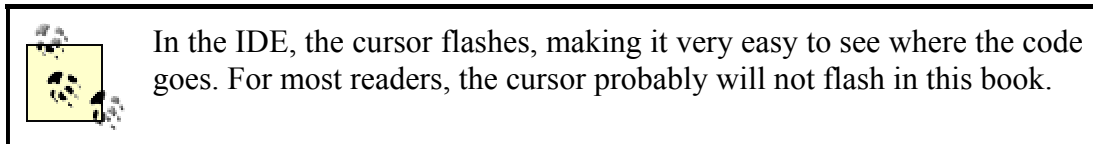
Any one of these steps is much easier than modifying these properties in code (though that is certainly still possible).

Once you have the form laid out the way you want, all that remains is to create an event handler for the Cancel button. Double-clicking the Cancel button will create the event handler, register it, and put you on the *code-behind* page (the page that holds the source code for this form), in which you can enter the event-handling logic, as shown in [Figure 13-6](#).

Figure 13-6. After double-clicking the Cancel button

The cursor is already in place; you have only to enter the one line of code:

```
Application.Exit( );
```



Visual Studio .NET generates all the code necessary to create and initialize the components. The complete source code is shown in [Example 13-2](#), including the one line of code you provided (shown in bold in this example) to handle the Cancel button-click event.

Example 13-2. Source code generated by the IDE

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace ProgCSharpWindowsForm
{
    /// <summary>
    /// Summary description for Form1.
    /// </summary>
    public class Form1 : System.Windows.Forms.Form
    {
        private System.Windows.Forms.Label lblOutput;
        private System.Windows.Forms.Button btnCancel;
```

```

/// <summary>
/// Required designer variable.
/// </summary>
private System.ComponentModel.Container components = null;

public Form1( )
{
    //
    // Required for Windows Form Designer support
    //
    InitializeComponent( );

    //
    // TODO: Add any constructor code
    // after InitializeComponent call
    //
}

/// <summary>
/// Clean up any resources being used.
/// </summary>
protected override void Dispose( bool disposing )
{
    if( disposing )
    {
        if (components != null)
        {
            components.Dispose( );
        }
    }
    base.Dispose( disposing );
}

#region Windows Form Designer generated code
/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
private void InitializeComponent( )
{
    this.lblOutput = new System.Windows.Forms.Label( );
    this.btnCancel = new System.Windows.Forms.Button( );
    this.SuspendLayout( );
    //
    // lblOutput
    //
    this.lblOutput.Font = new System.Drawing.Font("Arial", 15.75F,
        System.Drawing.FontStyle.Bold,
        System.Drawing.GraphicsUnit.Point, ((System.Byte)(0)));
    this.lblOutput.Location = new System.Drawing.Point(24, 16);
    this.lblOutput.Name = "lblOutput";
    this.lblOutput.Size = new System.Drawing.Size(136, 48);
    this.lblOutput.TabIndex = 0;
    this.lblOutput.Text = "Hello World";
    //
    // btnCancel
    //
    this.btnCancel.Location = new System.Drawing.Point(192, 208);
    this.btnCancel.Name = "btnCancel";
    this.btnCancel.TabIndex = 1;
    this.btnCancel.Text = "Cancel";
}

```

```

        this.btnCancel.Click += new System.EventHandler(
            this.btnCancel_Click);
        //
        // Form1
        //
        this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
        this.ClientSize = new System.Drawing.Size(292, 273);
        this.Controls.AddRange(new System.Windows.Forms.Control[] {
            this.btnCancel, this.lblOutput});
        this.Name = "Form1";
        this.Text = "Form1";
        this.ResumeLayout(false);
    }
    #endregion

    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    static void Main( )
    {
        Application.Run(new Form1( ));
    }

    private void btnCancel_Click(object sender, System.EventArgs e)
    {
        Application.Exit( );
    }
}
}

```



Some of the code in this listing has been reformatted to fit the printed page.

There is quite a bit of code in this listing that didn't appear in [Example 13-1](#), though most of it is not terribly important. When Visual Studio creates the application, it must add some boilerplate code that is not essential for this simple application.

A careful examination reveals that the essentials are the same, but there are some key differences worth examining. The listing starts with special comment marks:

```

/// <summary>
/// Summary description for Form1.
/// </summary>

```

These marks are used for creating documentation; they are explained in detail later in this chapter.

The form derives from `System.Windows.Forms.Form` as did our earlier example. The widgets are defined as in the previous example:

```
public class Form1 : System.Windows.Forms.Form
{
    private System.Windows.Forms.Label lblOutput;
    private System.Windows.Forms.Button btnCancel;
```

The designer creates a private container variable for its own use:

```
private System.ComponentModel.Container components = null;
```

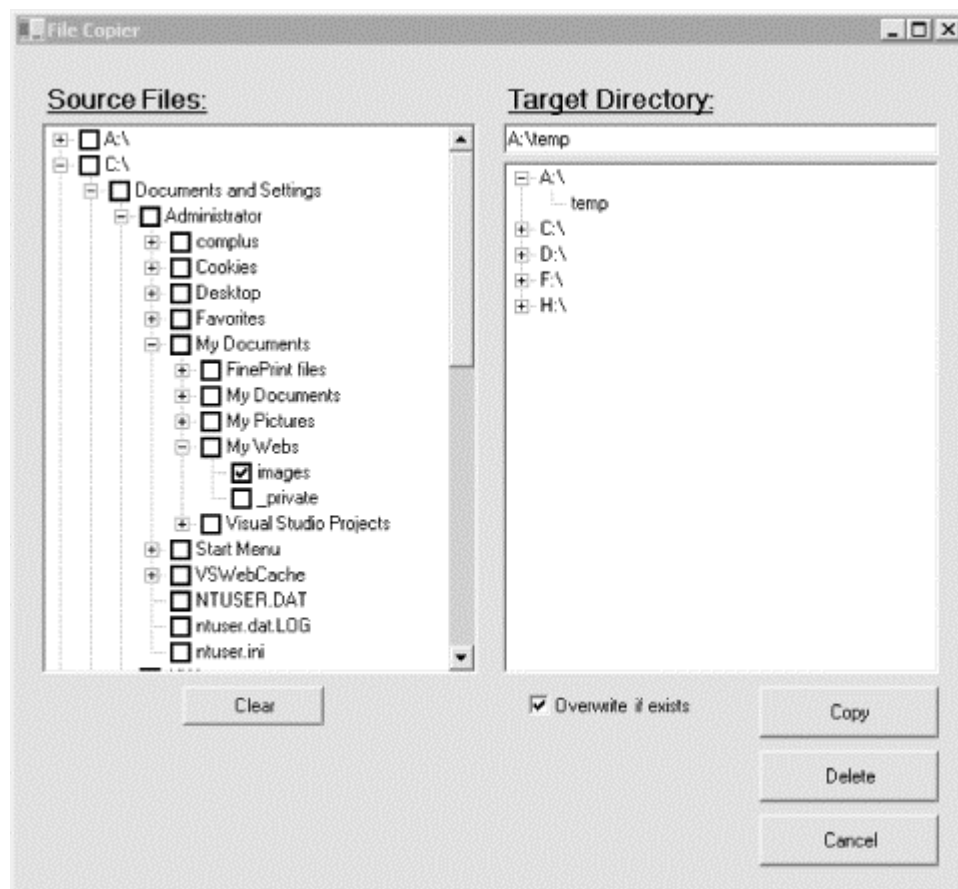
In this and in every Windows Form application generated by Visual Studio .NET, the constructor calls a private method, `InitializeComponent()`. This is used to define and set the properties of all the controls. The properties are set based on the values you've chosen (or on the default values you've left alone) in the designer. The `InitializeComponent()` method is marked with a comment that you should not modify the contents of this method; making changes to this method might confuse the designer.

This program will behave exactly as your earlier hand-crafted application did.

13.2 Creating a Windows Form Application

To see how Windows Forms can be used to create a more realistic Windows application, in this section you'll build a utility named `FileCopier` that copies all files from a group of directories selected by the user to a single target directory or device, such as a floppy or backup hard drive on the company network. Although you won't implement every possible feature, you can imagine programming this application so that you can mark dozens of files and have them copied to multiple disks, packing them as tightly as possible. You might even extend the application to compress the files. The true goal of this example is for you to exercise many of the C# skills learned in earlier chapters and to explore the `Windows.Forms` namespace.

For the purposes of this example and to keep the code simple, focus on the user interface and the steps needed to wire up its various controls. The final application UI is shown in [Figure 13-7](#).

Figure 13-7. The FileCopier user interface

The user interface for `FileCopier` consists of the following controls:

- Labels: Source Files and Target Directory
- Buttons: Clear, Copy, Delete, and Cancel
- An Overwrite if exists checkbox
- A text box displaying the path of the selected target directory
- Two large tree view controls, one for available source directories and one for available target devices and directories

The goal is to allow the user to check files (or entire directories) in the left tree view (source). If the user presses the Copy button, the files checked on the left side will be copied to the Target Directory specified in the right-hand control. If the user presses Delete, the checked files will be deleted.

The rest of this chapter implements a number of `FileCopier` features in order to demonstrate the fundamental features of Windows Forms.

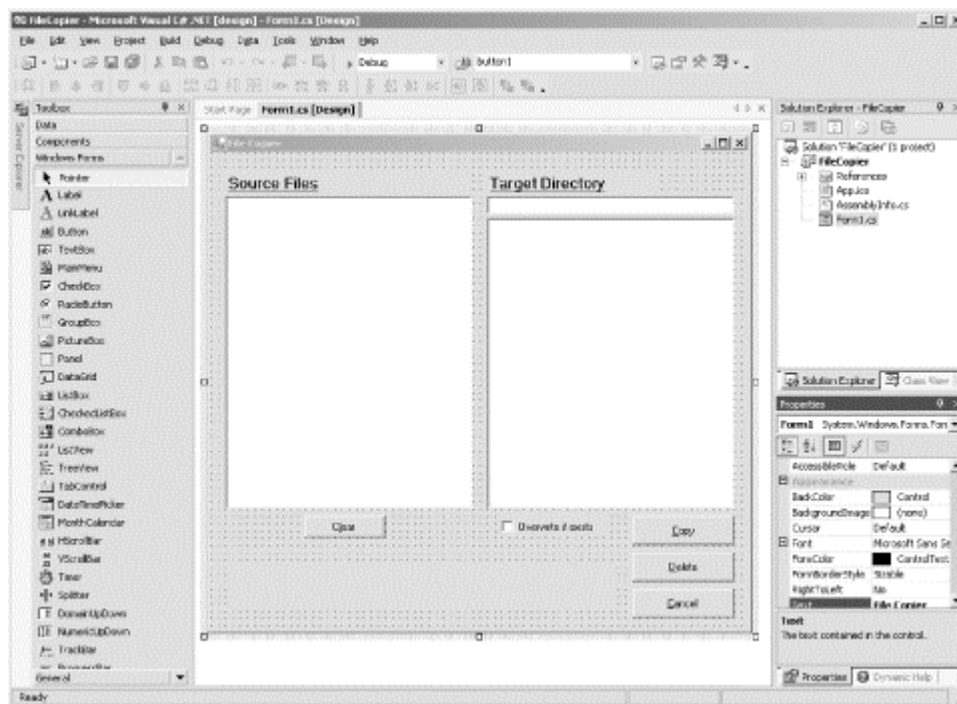
13.2.1 Creating the Basic UI Form

The first task is to open a new project named `FileCopier`. The IDE puts you into the Designer, in which you can drag widgets onto the form. You can expand the form to the size you want. Drag, drop, and set the Name properties of labels (`lblSource`, `lblTarget`, `lblStatus`), buttons (`btnClear`, `btnCopy`, `btnDelete`, `btnCancel`), a checkbox

(chkOverwrite), a textbox (txtTargetDir), and tree view controls (tvwSource, tvwTargetDir) from the Toolbox onto your form until it looks more or less like the one shown in [Figure 13-8](#).

You want checkboxes next to the directories and files in the source selection window but not in the target (where only one directory will be chosen). Set the `CheckBoxes` property on the left `TreeView` control, `tvwSource`, to `true`, and set the property on the right-hand `TreeView` control, `tvwTargetDir`, to `false`. To do so, click each control in turn and adjust the values in the Properties window.

Figure 13-8. Creating the form in the designer



Once this is done, double-click the Cancel button to create its event handler -- when you double-click a control, Visual Studio .NET creates an event handler for that object. One particular event is the target event, and Visual Studio .NET opens that event's event handler:

```
protected void btnCancel_Click (object sender, System.EventArgs e)
{
    Application.Exit( );
}
```

You can set many different events for the `TreeView` control. Do so programmatically by clicking the Events button in the Properties window. From there you can create new handlers, just by filling in a new event-handler method name. Visual Studio .NET will register the event handler and open the editor for the code, where it will create the header and put the cursor in an empty method body.

So much for the easy part. Visual Studio .NET will generate code to set up the form and initialize all the controls, but it won't fill the `TreeView` controls. That you must do by hand.

13.2.2 Populating the TreeView Controls

The two `TreeView` controls work identically, except that the left control, `tvwSource`, lists the directories and files, whereas the right control, `tvwTargetDir`, lists only directories. The `CheckBoxes` property on `tvwSource` is set to `true`, and on `tvwTargetDir` it is set to `false`. Also, although `tvwSource` will allow multiselect, which is the default for `TreeView` controls, you will enforce single selection for `tvwTargetDir`.

You'll factor the common code for both `TreeView` controls into a shared method `FillDirectoryTree` and pass in the control with a flag indicating whether to get the files. You'll call this method from the Form's constructor, once for each of the two controls:

```
FillDirectoryTree(tvwSource, true);
FillDirectoryTree(tvwTargetDir, false);
```

The `FillDirectoryTree` implementation names the `TreeView` parameter `tvw`. This will represent the source `TreeView` and the destination `TreeView` in turn. You'll need some classes from `System.IO`, so add a `using System.IO;` statement at the top of `Form1.cs`. Next, add the method declaration to `Form1.cs`:

```
private void FillDirectoryTree(TreeView tvw, bool isSource)
```

13.2.2.1 TreeNode objects

The `TreeView` control has a property, `Nodes`, which gets a `TreeNodeCollection` object. The `TreeNodeCollection` is a collection of `TreeNode` objects, each of which represents a node in the tree. Start by emptying that collection:

```
tvw.Nodes.Clear( );
```

You are ready to fill the `TreeView`'s `Nodes` collection by recursing through the directories of all the drives. First, get all the logical drives on the system. To do so, call a static method of the `Environment` object, `GetLogicalDrives( )`. The `Environment` class provides information about and access to the current platform environment. You can use the `Environment` object to get the machine name, OS version, system directory, and so forth, from the computer on which you are running your program.

```
string[] strDrives = Environment.GetLogicalDrives( );
```

`GetLogicalDrives( )` returns an array of strings, each of which represents the root directory of one of the logical drives. You will iterate over that collection, adding nodes to the `TreeView` control as you go.

```
foreach (string rootDirectoryName in strDrives)
{
```

You should process each drive within the `foreach` loop. You can add these two lines to limit the search to a particular drive (this is good if you have several large drives or some network drives):

```
if (rootDirectoryName != @"C:\")
    continue;
```

The very first thing you need to determine is whether the drive is ready. My hack for that is to get the list of top-level directories from the drive by calling `GetDirectories( )` on a `DirectoryInfo` object I created for the root directory:

```
DirectoryInfo dir = new DirectoryInfo(rootDirectoryName);
dir.GetDirectories( );
```

The `DirectoryInfo` class exposes instance methods for creating, moving, and enumerating through directories, their files, and their subdirectories. The `DirectoryInfo` class is covered in detail in [Chapter 21](#).

The `GetDirectories( )` method returns a list of directories, but throw this list away. You are calling it here only to generate an exception if the drive is not ready.

Wrap the call in a try block and take no action in the catch block. The effect is that if an exception is thrown, the drive is skipped. Once you know that the drive is ready, create a `TreeNode` to hold the root directory of the drive and add that node to the `TreeView` control:

```
TreeNode ndRoot = new TreeNode(rootDirectoryName);
tvw.Nodes.Add(ndRoot);
```

You now want to recurse through the directories, so you call into a new routine, `GetSubDirectoryNodes( )`, passing in the root node, the name of the root directory, and the flag indicating whether you want files:

```
if (isSource)
{
    GetSubDirectoryNodes(ndRoot, ndRoot.Text, true);
}
else
{
    GetSubDirectoryNodes(ndRoot, ndRoot.Text, false);
}
```

You are probably wondering why you need to pass in `ndRoot.Text` if you're already passing in `ndRoot`. Patience; you will see why this is needed when you recurse back into `GetSubDirectoryNodes`. You are now finished with `FillDirectoryTree( )`. See [Example 13-3](#) for a complete listing of this method.

13.2.2.2 Recursing through the subdirectories

`GetSubDirectoryNodes( )` begins by once again calling `GetDirectories( )`, this time stashing away the resulting array of `DirectoryInfo` objects:

```
private void GetSubDirectoryNodes(
    TreeNode parentNode, string fullName, bool getFileNames)
{
    DirectoryInfo dir = new DirectoryInfo(fullName);
    DirectoryInfo[] dirSubs = dir.GetDirectories( );
```

Notice that the node passed in is named `parentNode`. The current level of nodes will be considered children to the node passed in. This is how you map the directory structure to the hierarchy of the tree view.

Iterate over each subdirectory, skipping any that are marked `Hidden`:

```
foreach (DirectoryInfo dirSub in dirSubs)
{
    if ( (dirSub.Attributes &
        FileAttributes.Hidden) != 0 )
    {
        continue;
    }
}
```

`FileAttributes` is an enum; other possible values include `Archive`, `Compressed`, `Directory`, `Encrypted`, `Hidden`, `Normal`, `ReadOnly`, etc.



The property `dirSub.Attributes` is the bit pattern of the current attributes of the directory. If you logically AND that value with the bit pattern `FileAttributes.Hidden`, a bit is set if the file has the `hidden` attribute; otherwise all the bits are cleared. You can check for any hidden bit by testing whether the resulting `int` is other than zero.

Create a `TreeNode` with the directory name and add it to the `Nodes` collection of the node passed in to the method (`parentNode`):

```
TreeNode subNode = new TreeNode(dirSub.Name);
parentNode.Nodes.Add(subNode);
```

Now recurse back into the `GetSubDirectoryNodes( )` method, passing in the node you just created as the new parent, the full path as the full name of the parent, and the flag:

```
GetSubDirectoryNodes(subNode, dirSub.FullName, getFileNames);
```



Notice that the call to the `TreeNode` constructor uses the `Name` property of the `DirectoryInfo` object, while the call to `GetSubDirectoryNodes( )` uses the `FullName` property. If your directory is `c:\WinNT\Media\Sounds`, the `FullName` property will return the full path, while the `Name` property will return just `Sounds`. Pass in only the name to the node because that is what you want displayed in the tree view. Pass in the full name with path to the `GetSubDirectoryNodes( )` method so that the method can locate all the subdirectories on the disk. This answers the question asked earlier as to why you need to pass in the root node's name the first time you call this method; what is passed in is not the name of the node, it is the full path to the directory represented by the node!

13.2.2.3 Getting the files in the directory

Once you've recursed through the subdirectories, it is time to get the files for the directory if the `getFileNames` flag is `true`. To do so, call the `GetFiles( )` method on the `DirectoryInfo` object. An array of `FileInfo` objects is returned:

```
if (getFileNames)
{
    // Get any files for this node.
    FileInfo[] files = dir.GetFiles( );
```

The `FileInfo` class (covered in [Chapter 21](#)) provides instance methods for manipulating files.

You can now iterate over this collection, accessing the `Name` property of the `FileInfo` object and passing that name to the constructor of a `TreeNode`, which you then add to the parent node's `Nodes` collection (thus creating a child node). There is no recursion this time because files do not have subdirectories:

```
foreach (FileInfo file in files)
{
    TreeNode fileNode = new TreeNode(file.Name);
    parentNode.Nodes.Add(fileNode);
}
```

That's all it takes to fill the two tree views. See [Example 13-3](#) for a complete listing of this method.



If you found any of this confusing, I highly recommend putting the code into your debugger and stepping through the recursion; you can watch the `TreeView` build its nodes.

13.2.3 Handling TreeView Events

You must handle a number of events in this example. First, the user might click `Cancel`, `Copy`, `Clear`, or `Delete`. Second, the user might click one of the checkboxes in the left `TreeView` or one of the nodes in the right `TreeView`.

Let's consider the clicks on the `TreeViews` first, as they are the more interesting, and potentially the more challenging.

13.2.3.1 Clicking the source TreeView

There are two `TreeView` objects, each with its own event handler. Consider the source `TreeView` object first. The user checks the files and directories he wants to copy from. Each time the user clicks a file or directory, a number of events are raised. The event you must handle is `AfterCheck`.

To do so, implement a custom event-handler method you will create and name `tvwSource_AfterCheck( )`. Visual Studio .NET will wire this to the event handler, or if you are not using the integrated development environment, you must do so yourself.

```
tvwSource.AfterCheck +=
new System.Windows.Forms.TreeViewEventHandler
(this.tvwSource_AfterCheck);
```

The implementation of `AfterCheck( )` delegates the work to a recursible method named `SetCheck( )` that you'll also write. To add the `AfterCheck` event, select the `tvwSource` control, click the Events icon in the Properties window, then double-click on `AfterCheck`. This will add the event, wire it up, and place you in the code editor where you can add the body of the method:

```
private void tvwSource_AfterCheck (
object sender, System.Windows.Forms.TreeViewEventArgs e)
{
    SetCheck(e.Node,e.Node.Checked);
}
```

The event handler passes in the `sender` object and an object of type `TreeViewEventArgs`. It turns out that you can get the node from this `TreeViewEventArgs` object (`e`). Call `SetCheck( )`, passing in the node and the state of whether the node has been checked.

Each node has a `Nodes` property, which gets a `TreeNodeCollection` containing all the subnodes. `SetCheck( )` recurses through the current node's `Nodes` collection, setting each subnode's check mark to match that of the node that was checked. In other words, when you check a directory, all its files and subdirectories are checked, recursively, all the way down.

It's Turtles, All the Way Down

Here's my favorite story on recursion: it happened that a famous Darwinist was telling a story about primitive creation myths. "Some peoples," he said, "believe the world rests on the back of a great turtle. Of course, that raises the question: on what does the turtle rest?"

An elderly woman from the back of the room stood up and said, "Very clever, Sonny, but it's turtles, all the way down."

For each `TreeNode` in the `Nodes` collection, check to see if it is a leaf. A node is a leaf if its own `Nodes` collection has a count of zero. If it is a leaf, set its check property to whatever was passed in as a parameter. If it is not a leaf, recurse.

```
private void SetCheck(TreeNode node, bool check)
{
    // find all the child nodes from this node
    foreach (TreeNode n in node.Nodes)
    {
        n.Checked = check;    // check the node

        // if this is a node in the tree, recurse
        if (n.Nodes.Count != 0)
        {
            SetCheck(n,check);
        }
    }
}
```

This propagates the check mark (or clears the check mark) down through the entire structure. In this way, the user can indicate that he wants to select all the files in all the subdirectories by clicking a single directory.

13.2.3.2 Clicking the target TreeView

The event handler for the target `TreeView` is somewhat trickier. The event itself is `AfterSelect`. (Remember that the target `TreeView` does not have checkboxes.) This time, you want to take the one directory chosen and put its full path into the text box at the upper-left corner of the form.

To do so, you must work your way up through the nodes, finding the name of each parent directory and building the full path:

```
private void tvwTargetDir_AfterSelect (
    object sender, System.Windows.Forms.TreeViewEventArgs e)
{
    string theFullPath = GetParentString(e.Node);
```

We'll look at `GetParentString( )` in just a moment. Once you have the full path, you must lop off the backslash (if any) on the end and then you can fill the text box:

```
if (theFullPath.EndsWith("\\"))
{
    theFullPath =
        theFullPath.Substring(0,theFullPath.Length-1);
}

txtTargetDir.Text = theFullPath;
```

The `GetParentString( )` method takes a node and returns a string with the full path. To do so, it recurses upward through the path, adding the backslash after any node that is not a leaf:

```
private string GetParentString(TreeNode node)
{
    if(node.Parent == null)
    {
        return node.Text;
    }
    else
    {
        return GetParentString(node.Parent) + node.Text +
            (node.Nodes.Count == 0 ? "" : "\\");
    }
}
```



The conditional operator (?) is the only ternary operator in C# (a ternary operator takes three terms). The logic is "test whether `node.Nodes.Count` is zero; if so return the value before the colon (in this case an empty string). Otherwise return the value after the colon (in this case a backslash)."

The recursion stops when there is no parent; that is, when you hit the root directory.

13.2.3.3 Handling the Clear button event

Given the `SetCheck( )` method developed earlier, handling the Clear button's click event is trivial:

```
protected void btnClear_Click (object sender, System.EventArgs e)
{
    foreach (TreeNode node in tvwSource.Nodes)
    {
        SetCheck(node, false);
    }
}
```

Just call the `SetCheck( )` method on the root nodes and tell them to recursively uncheck all their contained nodes.

13.2.4 Implementing the Copy Button Event

Now that you can check the files and pick the target directory, you're ready to handle the Copy button-click event. The very first thing you need to do is to get a list of which files were selected. What you want is an array of `FileInfo` objects, but you have no idea how many objects will be in the list. That is a perfect job for `ArrayList`. Delegate responsibility for filling the list to a method called `GetFileList( )`:

```
private void btnCopy_Click (
    object sender, System.EventArgs e)
{
    ArrayList fileList = GetFileList( );
}
```

Let's pick that method apart before returning to the event handler.

13.2.4.1 Getting the selected files

Start by instantiating a new `ArrayList` object to hold the strings representing the names of all the files selected:

```
private ArrayList GetFileList( )
{
    ArrayList fileNames = new ArrayList( );
}
```

To get the selected filenames, you can walk through the source `TreeView` control:

```
foreach (TreeNode theNode in tvwSource.Nodes)
{
    GetCheckedFiles(theNode, fileNames);
}
```

To see how this works, step into the `GetCheckedFiles( )` method. This method is pretty simple: it examines the node it was handed. If that node has no children (`node.Nodes.Count == 0`), it is a leaf. If that leaf is checked, get the full path (by calling `GetParentString( )` on the node) and add it to the `ArrayList` passed in as a parameter:

```
private void GetCheckedFiles(TreeNode node, ArrayList fileNames)
{
    if (node.Nodes.Count == 0)
    {
        if (node.Checked)
        {
            string fullPath = GetParentString(node);
            fileNames.Add(fullPath);
        }
    }
}
```

If the node is *not* a leaf, recurse down the tree, finding the child nodes:

```
else
{
    foreach (TreeNode n in node.Nodes)
    {
        GetCheckedFiles(n, fileNames);
    }
}
```

This will return the `ArrayList` filled with all the filenames. Back in `GetFileList( )`, use this `ArrayList` of filenames to create a second `ArrayList`, this time to hold the actual `FileInfo` objects:

```
ArrayList fileList = new ArrayList( );
```

Notice that once again you do not tell the `ArrayList` constructor what kind of object it will hold. This is one of the advantages of a rooted type-system: the collection only needs to know that it has some kind of `Object`; because all types are derived from `Object`, the list can hold `FileInfo` objects as easily as it can hold `string` objects.

You can now iterate through the filenames in `ArrayList`, picking out each name and instantiating a `FileInfo` object with it. You can detect if it is a file or a directory by calling the `Exists` property, which will return `false` if the `File` object you created is actually a directory. If it is a `File`, you can add it to the new `ArrayList`:

```
foreach (string fileName in fileNames)
{
    FileInfo file = new FileInfo(fileName);

    if (file.Exists)
    {
        fileList.Add(file);
    }
}
```

13.2.4.2 Sorting the list of selected files

You want to work your way through the list of selected files in large to small order so that you can pack the target disk as tightly as possible. You must therefore sort the `ArrayList`. You can call its `Sort( )` method, but how will it know how to sort `File` objects? Remember, the `ArrayList` has no special knowledge about its contents.

To solve this, you must pass in an `IComparer` interface. We'll create a class called `FileComparer` that will implement this interface and that will know how to sort `FileInfo` objects:

```
public class FileComparer : IComparer
{
```

This class has only one method, `Compare( )`, which takes two objects as arguments:

```
public int Compare (object f1, object f2)
{
```

The normal approach is to return 1 if the first object (`f1`) is larger than the second (`f2`), to return -1 if the opposite is true, and to return 0 if they are equal. In this case, however, you want the list sorted from big to small, so you should reverse the return values.



Since this is the only use of this `compare` method, it is reasonable to put this special knowledge that the sort is from big to small right into the `compare` method itself. The alternative is to sort small to big, and have the calling method reverse the results, as you saw in [Example 12-1](#).

To test the length of the `FileInfo` object, you must cast the `Object` parameters to `FileInfo` objects (which is safe, as you know this method will never receive anything else):

```
    FileInfo file1 = (FileInfo) f1;
    FileInfo file2 = (FileInfo) f2;
    if (file1.Length > file2.Length)
    {
        return -1;
    }
    if (file1.Length < file2.Length)
    {
        return 1;
    }
    return 0;
}
```



In a production program, you might want to test the type of the object and perhaps handle the exception if the object is not of the expected type.

Returning to `GetFileList( )`, you were about to instantiate the `IComparer` reference and pass it to the `Sort( )` method of `fileList`:

```
IComparer comparer = (IComparer) new FileComparer( );
fileList.Sort(comparer);
```

That done, you can return `fileList` to the calling method:

```
return fileList;
```

The calling method was `btnCopy_Click`. Remember you went off to `GetFileList( )` in the first line of the event handler!

```
protected void btnCopy_Click (object sender, System.EventArgs e)
{
    ArrayList fileList = GetFileList( );
```

At this point you've returned with a sorted list of `File` objects, each representing a file selected in the source `TreeView`.

You can now iterate through the list, copying the files and updating the UI:

```
foreach (FileInfo file in fileList)
{
    try
    {
        lblStatus.Text = "Copying " +
            txtTargetDir.Text + "\\\" +
            file.Name + "...";
        Application.DoEvents( );

        file.CopyTo(txtTargetDir.Text + "\\\" +
            file.Name, chkOverwrite.Checked);
    }

    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}
lblStatus.Text = "Done.";
Application.DoEvents( );
```

As you go, write the progress to the `lblStatus` label and call `Application.DoEvents( )` to give the UI an opportunity to redraw. Then call `CopyTo( )` on the file, passing in the target directory obtained from the text field, and a Boolean flag indicating whether the file should be overwritten if it already exists.

You'll notice that the flag you pass in is the value of the `chkOverWrite` checkbox. The `Checked` property evaluates `true` if the checkbox is checked and `false` if not.

The copy is wrapped in a `try` block because you can anticipate any number of things going wrong when copying files. For now, handle all exceptions by popping up a dialog box with the error, but you might want to take corrective action in a commercial application.

That's it; you've implemented file copying!

13.2.5 Handling the Delete Button Event

The code to handle the delete event is even simpler. The very first thing you do is ask the user if she is sure she wants to delete the files:

```
protected void btnDelete_Click
(object sender, System.EventArgs e)
{
    System.Windows.Forms.DialogResult result =
        MessageBox.Show(
            "Are you quite sure?",           // msg
            "Delete Files",                 // caption
            MessageBoxButtons.OKCancel,      // buttons
            MessageBoxIcon.Exclamation,      // icons
            MessageBoxDefaultButton.Button2); // default button
}
```

You can use the `MessageBox` static `Show( )` method, passing in the message you want to display, the title "Delete Files" as a string, and flags.

- `MessageBox.OKCancel` asks for two buttons: OK and Cancel.
- `MessageBox.IconExclamation` indicates that you want to display an exclamation mark icon.
- `MessageBox.DefaultButton.Button2` sets the second button (Cancel) as the default choice.

When the user chooses OK or Cancel, the result is passed back as a `System.Windows.Forms.DialogResult` enumerated value. You can test this value to see if the user pressed OK:

```
if (result == System.Windows.Forms.DialogResult.OK)
{
    // ...
}
```

If so, you can get the list of `fileNames` and iterate through it, deleting each as you go:

```
ArrayList fileNames = GetFileList( );

foreach (FileInfo file in fileNames)
{
    try
    {
        lblStatus.Text = "Deleting " +
            txtTargetDir.Text + "\\\" +
            file.Name + "...";
        Application.DoEvents( );

        file.Delete( );
    }

    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}

lblStatus.Text = "Done.";
Application.DoEvents( );
```

This code is identical to the copy code, except that the method that is called on the file is `Delete( )`.

[Example 13-3](#) provides the commented source code for this example.



To save space, this example shows only the custom methods and leaves out the declarations of the `Windows.Forms` objects as well as the boilerplate code produced by Visual Studio .NET. As explained in the preface, you can download the complete source code from my web site, <http://www.libertyassociates.com/>.

Example 13-3. File copier source code

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.IO;

/// <remarks>
///     File Copier - WinForms demonstration program
///     (c) Copyright 2001 Liberty Associates, Inc.
/// </remarks>
namespace FileCopier
{
    /// <summary>
    /// Form demonstrating Windows Forms implementation
    /// </summary>
    public class Form1 : System.Windows.Forms.Form
    {

        // < declarations of Windows widgets cut here >

        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.Container components = null;

        /// <summary>
        ///     internal class which knows how to compare
        ///     two files we want to sort large to small,
        ///     so reverse the normal return values.
        /// </summary>
        public class FileComparer : IComparer
        {
            public int Compare (object f1, object f2)
            {
                FileInfo file1 = (FileInfo) f1;
                FileInfo file2 = (FileInfo) f2;
                if (file1.Length > file2.Length)
                {
                    return -1;
                }
                if (file1.Length < file2.Length)
                {
                    return 1;
                }
                return 0;
            }
        }
    }
}
```

```

public Form1( )
{
    //
    // Required for Windows Form Designer support
    //
    InitializeComponent( );

    // fill the source and target directory trees
    FillDirectoryTree(tvwSource, true);
    FillDirectoryTree(tvwTargetDir, false);
}

/// <summary>
/// Fill the directory tree for either the Source or
/// Target TreeView.
/// </summary>
private void FillDirectoryTree(
    TreeView tvw, bool isSource)
{
    // Populate tvwSource, the Source TreeView,
    // with the contents of
    // the local hard drive.
    // First clear all the nodes.
    tvw.Nodes.Clear( );

    // Get the logical drives and put them into the
    // root nodes. Fill an array with all the
    // logical drives on the machine.
    string[] strDrives =
        Environment.GetLogicalDrives( );

    // Iterate through the drives, adding them to the tree.
    // Use a try/catch block, so if a drive is not ready,
    // e.g. an empty floppy or CD,
    // it will not be added to the tree.
    foreach (string rootDirectoryName in strDrives)
    {
        if (rootDirectoryName != @"C:\")
            continue;
        try
        {
            // Fill an array with all the first level
            // subdirectories. If the drive is
            // not ready, this will throw an exception.
            DirectoryInfo dir =
                new DirectoryInfo(rootDirectoryName);
            dir.GetDirectories( );

            TreeNode ndRoot = new TreeNode(rootDirectoryName);

            // Add a node for each root directory.
            tvw.Nodes.Add(ndRoot);
        }
    }
}

```

```

        // Add subdirectory nodes.
        // If Treeview is the source,
        // then also get the filenames.
        if (isSource)
        {
            GetSubDirectoryNodes(
                ndRoot, ndRoot.Text, true);
        }
        else
        {
            GetSubDirectoryNodes(
                ndRoot, ndRoot.Text, false);
        }
    }
    // Catch any errors such as
    // Drive not ready.
    catch (Exception e)
    {
        MessageBox.Show(e.Message);
    }
}
} // close for FillSourceDirectoryTree

/// <summary>
/// Gets all the subdirectories below the
/// passed in directory node.
/// Adds to the directory tree.
/// The parameters passed in are the parent node
/// for this subdirectory,
/// the full path name of this subdirectory,
/// and a Boolean to indicate
/// whether or not to get the files in the subdirectory.
/// </summary>
private void GetSubDirectoryNodes(
    TreeNode parentNode, string fullName, bool getFileNames)
{
    DirectoryInfo dir = new DirectoryInfo(fullName);
    DirectoryInfo[] dirSubs = dir.GetDirectories();

    // Add a child node for each subdirectory.
    foreach (DirectoryInfo dirSub in dirSubs)
    {
        // do not show hidden folders
        if ( (dirSub.Attributes & FileAttributes.Hidden)
            != 0 )
        {
            continue;
        }

        /// <summary>
        /// Each directory contains the full path.
        /// We need to split it on the backslashes,
        /// and only use
        /// the last node in the tree.
        /// Need to double the backslash since it
        /// is normally
        /// an escape character
        /// </summary>
        TreeNode subNode = new TreeNode(dirSub.Name);
        parentNode.Nodes.Add(subNode);
    }
}

```

```

        // Call GetSubDirectoryNodes recursively.
        GetSubDirectoryNodes(
            subNode, dirSub.FullName, getFileNames);
    }
    if (getFileNames)
    {
        // Get any files for this node.
        FileInfo[] files = dir.GetFiles( );

        // After placing the nodes,
        // now place the files in that subdirectory.
        foreach (FileInfo file in files)
        {
            TreeNode fileNode = new TreeNode(file.Name);
            parentNode.Nodes.Add(fileNode);
        }
    }
}

// < boilerplate code cut here >

/// <summary>
/// The main entry point for the application.
/// </summary>
[STAThread]
static void Main( )
{
    Application.Run(new Form1( ));
}

/// <summary>
/// Create an ordered list of all
/// the selected files, copy to the
/// target directory
/// </summary>
private void btnCopy_Click(object sender,
    System.EventArgs e)
{
    // get the list
    ArrayList fileList = GetFileList( );

    // copy the files
    foreach (FileInfo file in fileList)
    {
        try
        {
            // update the label to show progress
            lblStatus.Text = "Copying " + txtTargetDir.Text +
                "\\\" + file.Name + "...";
            Application.DoEvents( );

            // copy the file to its destination location
            file.CopyTo(txtTargetDir.Text + "\\\" +
                file.Name, chkOverwrite.Checked);
        }
    }
}

```

```

        catch (Exception ex)
        {
            // you may want to do more than
            // just show the message
            MessageBox.Show(ex.Message);
        }
    }
    lblStatus.Text = "Done.";
    Application.DoEvents( );
}

/// <summary>
///     on cancel, exit
/// </summary>
private void btnCancel_Click(object sender, System.EventArgs e)
{
    Application.Exit( );
}

/// <summary>
///     Tell the root of each tree to uncheck
///     all the nodes below
/// </summary>
private void btnClear_Click(object sender, System.EventArgs e)
{
    // get the top most node for each drive
    // and tell it to clear recursively
    foreach (TreeNode node in tvwSource.Nodes)
    {
        SetCheck(node, false);
    }
}

/// <summary>
///     check that the user does want to delete
///     Make a list and delete each in turn
/// </summary>
private void btnDelete_Click(object sender, System.EventArgs e)
{
    // ask them if they are sure
    System.Windows.Forms.DialogResult result =
        MessageBox.Show(
            "Are you quite sure?",           // msg
            "Delete Files",                  // caption
            MessageBoxButtons.OKCancel,      // buttons
            MessageBoxIcon.Exclamation,      // icons
            MessageBoxDefaultButton.Button2); // default button

    // if they are sure...
    if (result == System.Windows.Forms.DialogResult.OK)
    {
        // iterate through the list and delete them.
        // get the list of selected files
        ArrayList fileNames = GetFileList( );
    }
}

```

```

        foreach (FileInfo file in fileNames)
        {
            try
            {
                // update the label to show progress
                lblStatus.Text = "Deleting " +
                    txtTargetDir.Text + "\\\" +
                    file.Name + "...";
                Application.DoEvents( );

                // Danger Will Robinson!
                file.Delete( );
            }

            catch (Exception ex)
            {
                // you may want to do more than
                // just show the message
                MessageBox.Show(ex.Message);
            }
        }
        lblStatus.Text = "Done.";
        Application.DoEvents( );
    }
}

/// <summary>
///     Get the full path of the chosen directory
///     copy it to txtTargetDir
/// </summary>
private void tvwTargetDir_AfterSelect(
    object sender,
    System.Windows.Forms.TreeViewEventArgs e)
{
    // get the full path for the selected directory
    string theFullPath = GetParentString(e.Node);

    // if it is not a leaf, it will end with a back slash
    // remove the backslash
    if (theFullPath.EndsWith("\\\"))
    {
        theFullPath =
            theFullPath.Substring(0,theFullPath.Length-1);
    }
    // insert the path in the text box
    txtTargetDir.Text = theFullPath;
}

/// <summary>
///     Mark each node below the current
///     one with the current value of checked
/// </summary>
private void tvwSource_AfterCheck(object sender,
    System.Windows.Forms.TreeViewEventArgs e)
{
    // Call a recursive method.
    // e.node is the node which was checked by the user.
    // The state of the check mark is already
    // changed by the time you get here.
    // Therefore, we want to pass along

```

```

        // the state of e.node.Checked.
        SetCheck(e.Node,e.Node.Checked);
    }

    /// <summary>
    ///     recursively set or clear check marks
    /// </summary>
    private void SetCheck(TreeNode node, bool check)
    {
        // find all the child nodes from this node
        foreach (TreeNode n in node.Nodes)
        {
            n.Checked = check;    // check the node

            // if this is a node in the tree, recurse
            if (n.Nodes.Count != 0)
            {
                SetCheck(n,check);
            }
        }
    }

    /// <summary>
    ///     Given a node and an array list
    ///     fill the list with the names of
    ///     all the checked files
    /// </summary>
    // Fill the ArrayList with the full paths of
    // all the files checked
    private void GetCheckedFiles(TreeNode node,
        ArrayList fileNames)
    {
        // if this is a leaf...
        if (node.Nodes.Count == 0)
        {
            // if the node was checked...
            if (node.Checked)
            {
                // get the full path and add it to the arrayList
                string fullPath = GetParentString(node);
                fileNames.Add(fullPath);
            }
        }
        else // if this node is not a leaf
        {
            // if this node is not a leaf
            foreach (TreeNode n in node.Nodes)
            {
                GetCheckedFiles(n,fileNames);
            }
        }
    }
}

```

```

/// <summary>
///     Given a node, return the
///     full path name
/// </summary>
private string GetParentString(TreeNode node)
{
    // if this is the root node (c:\) return the text
    if (node.Parent == null)
    {
        return node.Text;
    }
    else
    {
        // recurse up and get the path then
        // add this node and a slash
        // if this node is the leaf, don't add the slash
        return GetParentString(node.Parent) + node.Text +
            (node.Nodes.Count == 0 ? "" : "\\");
    }
}

/// <summary>
///     shared by delete and copy
///     creates an ordered list of all
///     the selected files
/// </summary>
private ArrayList GetFileList( )
{
    // create an unsorted array list of the full file names
    ArrayList fileNames = new ArrayList( );

    // fill the fileNames ArrayList with the
    // full path of each file to copy
    foreach (TreeNode theNode in tvwSource.Nodes)
    {
        GetCheckedFiles(theNode, fileNames);
    }

    // Create a list to hold the FileInfo objects
    ArrayList fileList = new ArrayList( );

    // for each of the file names we have in our unsorted list
    // if the name corresponds to a file (and not a directory)
    // add it to the file list
    foreach (string fileName in fileNames)
    {
        // create a file with the name
        FileInfo file = new FileInfo(fileName);

        // see if it exists on the disk
        // this fails if it was a directory
        if (file.Exists)
        {
            // both the key and the value are the file
            // would it be easier to have an empty value?
            fileList.Add(file);
        }
    }
}

```