Chapter 1

Creating Your First C# Windows Program

In This Chapter

- What's a program? What is C#? Where am I?
- Creating a Windows program
- Making sure your Visual Studio 2005 C# is in tune

In this chapter, I explain a little bit about computers, computer languages, C#, and Visual Studio 2005. Then, I take you through the steps for creating a very simple Windows program written in C#.

Getting a Handle on Computer Languages, C#, and .NET

A computer is an amazingly fast, but incredibly stupid servant. Computers will do anything you ask them to (within reason), and they do it extremely fast — and they're getting faster all the time. As of this writing, the common PC processing chip can handle well over a billion instructions per second. That's *billion,* with a "b."

Unfortunately, computers don't understand anything that resembles a human language. Oh, you may come back at me and say something like, "Hey, my telephone lets me dial my friend by just speaking his name. I know that a tiny computer runs my telephone. So that computer speaks English." But it's a computer program that understands English, not the computer itself.

The language that computers understand is often called *machine language.* It is possible, but extremely difficult and error prone, for humans to write machine language.

For historical reasons, machine language is also known as assembly language. In the old days, each manufacturer provided a program called an assembler that would convert special words into individual machine instructions. Thus, you might write something really cryptic like `MOV AX,CX`. (That's an actual Intel processor instruction, by the way.) The assembler would convert that instruction into a pattern of bits corresponding to a single machine instruction.

Humans and computers have decided to meet somewhere in the middle. Programmers create their programs in a language that is not nearly as free as human speech but a lot more flexible and easy to use than machine language. The languages that occupy this middle ground — C#, for example — are called *high-level* computer languages. (*High* is a relative term here.)

What's a program?

What is a program? In one sense, a Windows program is an executable file that you can run by double-clicking its icon. For example, the version of Microsoft Word that I'm using to write this book is a program. You call that an *executable program,* or *executable* for short. The names of executable program files generally end with the extension `.exe`.

But a program is something else, as well. An executable program consists of one or more *source files.* A C# program file is a text file that contains a sequence of C# commands, which fit together according to the laws of C# grammar. This file is known as a *source file,* probably because it's a source of frustration and anxiety.

What's C#?

The C# programming language is one of those intermediate languages that programmers use to create executable programs. C# fills the gap between the powerful-but-complicated C++ and the easy-to-use-but-limited Visual Basic — well, versions 6.0 and earlier, anyway. (Visual Basic's newer .NET incarnation is almost on par with C# in most respects. As the flagship language of .NET, C# tends to introduce most new features first.) A C# program file carries the extension `.CS`.

Some wags have pointed out that C-sharp and D-flat are the same note, but you should not refer to this new language as D-flat within earshot of Redmond, Washington.

C# is

- **Flexible:** C# programs can execute on the current machine, or they can be transmitted over the Web and executed on some distant computer.
- **Powerful:** C# has essentially the same command set as C++, but with the rough edges filed smooth.
- **Easier to use:** C# modifies the commands responsible for most C++ errors so you spend far less time chasing down those errors.
- **Visually oriented:** The .NET code library that C# uses for many of its capabilities provides the help needed to readily create complicated display frames with drop-down lists, tabbed windows, grouped buttons, scroll bars, and background images, to name just a few.
- **Internet friendly:** C# plays a pivotal role in the .NET Framework, Microsoft's current approach to programming for Windows, the Internet, and beyond. .NET is pronounced *dot net*.
- **Secure:** Any language intended for use on the Internet must include serious security to protect against malevolent hackers.

Finally, C# is an integral part of .NET.

What's .NET?

.NET began a few years ago as Microsoft's strategy to open up the Web to mere mortals like you and me. Today it's bigger than that, encompassing everything Microsoft does. In particular, it's the new way to program for Windows. It also gives a C-based language, C#, the simple, visual tools that made Visual Basic so popular. A little background will help you see the roots of C# and .NET.

Internet programming was traditionally very difficult in older languages like C and C++. Sun Microsystems responded to that problem by creating the Java programming language. To create Java, Sun took the grammar of C++, made it a lot more user friendly, and centered it around distributed development.

When programmers say "distributed," they're describing geographically dispersed computers running programs that talk to each other — in many cases, via the Internet.

When Microsoft licensed Java some years ago, it ran into legal difficulties with Sun over changes it wanted to make to the language. As a result, Microsoft more or less gave up on Java and started looking for ways to compete with it.

Being forced out of Java was just as well because Java has a serious problem: Although Java is a capable language, you pretty much have to write your entire program in Java to get its full benefit. Microsoft had too many developers and too many millions of lines of existing source code, so Microsoft had to come up with some way to support multiple languages. Enter .NET.

.NET is a framework, in many ways similar to Java's libraries, because the C# language is highly similar to the Java language. Just as *Java* is both the language itself and its extensive code library, *C#* is really much more than just the keywords and syntax of the C# language. It's those things empowered by a thoroughly object-oriented library containing thousands of code elements that simplify doing about any kind of programming you can imagine, from Web-based databases to cryptography to the humble Windows dialog box.

The previous generation platform was made up of a hodgepodge of tools with cryptic names. .NET updates all that with Visual Studio 2005, with more focused .NET versions of its Web and database technologies, newer versions of Windows, and .NET-enabled servers. .NET supports emerging communication standards such as XML and SOAP rather than Microsoft's proprietary formats. Finally, .NET supports the hottest buzzwords since *object-oriented*: Web Services.

Microsoft would claim that .NET is much superior to Sun's suite of Web tools based on Java, but that's not the point. Unlike Java, .NET does not require you to rewrite existing programs. A Visual Basic programmer can add just a few lines to make an existing program "Web knowledgeable" (meaning that it knows how to get data off the Internet). .NET supports all the common Microsoft languages and more than 40 other languages written by third-party vendors (see `www.gotdotnet.com/team/lang` for the latest list). However, C# is the flagship language of the .NET fleet. C# is always the first language to access every new feature of .NET.

What is Visual Studio 2005? What about Visual C#?

You sure ask lots of questions. The first "Visual" language from Microsoft was Visual Basic, code-named "Thunder." The first popular C-based programming language from Microsoft was Visual C++. Like Visual Basic, it was called "Visual" because it had a built-in graphical user interface (GUI — pronounced *gooey*). This GUI included everything you needed to develop nifty-giffy C++ programs.

Eventually, Microsoft rolled all its languages into a single environment — Visual Studio. As Visual Studio 6.0 started getting a little long in the tooth, developers anxiously awaited Version 7. Shortly before its release, however, Microsoft decided to rename it Visual Studio .NET to highlight this new environment's relationship to .NET.

That sounded like a marketing ploy to me until I started delving into it. Visual Studio .NET differed quite a bit from its predecessors — enough so to warrant a new name. Visual Studio 2005 is the successor to the original Visual Studio .NET. (See Bonus Chapter 4 on the CD for a tour of some of Visual Studio's more potent features.)

Microsoft calls its implementation of the language Visual C#. In reality, Visual C# is nothing more than the C# component of Visual Studio. C# is C#, with or without the Visual Studio.

Okay, that's it. No more questions.

Creating a Windows Application with C#

To help you get your feet wet with C# and Visual Studio, this section takes you through the steps for creating a simple Windows program. Windows programs are commonly called Windows applications, WinApps or WinForms apps for short.

Because this book focuses on the C# language, it's not a Web-programming book, a database book, or a Windows programming book per se. In particular, this chapter constitutes the only coverage of Windows Forms visual programming. All I have room to do is give you this small taste.

In addition to introducing Windows Forms, this program serves as a test of your Visual Studio environment. This is a test; this is only a test. Had it been an actual Windows program . . . Wait, it *is* an actual Windows program. If you can successfully create, build, and execute this program, your Visual Studio environment is set up properly, and you're ready to rock.

Creating the template

Writing Windows applications from scratch is a notoriously difficult process. With numerous session handles, descriptors, and contexts, creating even a simple Windows program poses innumerable challenges.

Visual Studio 2005 in general and C# in particular greatly simplify the task of creating your basic WinApp. To be honest, I'm a little disappointed that you don't get to go through the thrill of doing it by hand. In fact, why not switch over to Visual C++ and . . . okay, bad idea.

Because Visual C# is built specifically to execute under Windows, it can shield you from many of the complexities of writing Windows programs from scratch. In addition, Visual Studio 2005 includes an Applications Wizard that builds template programs.

Typically, *template programs* don't actually do anything — at least, not anything useful (sounds like most of my programs). However, they do get you beyond that initial hurdle of getting started. Some template programs are reasonably sophisticated. In fact, you'll be amazed at how much capability the App Wizard can build on its own.

After you've completed the Visual Studio 2005 installation, follow these steps to create the template:

1. **To start Visual Studio, choose Start➪All Programs➪Microsoft Visual Studio 2005➪Microsoft Visual Studio 2005, as shown in Figure 1-1.**

   After some gnashing of CPU teeth and thrashing of disk, the Visual Studio desktop appears. Now things are getting interesting.

2. **Choose File➪New➪Project, as shown in Figure 1-2.**

   Visual Studio responds by opening the New Project dialog box, as shown in Figure 1-3.

   A *project* is a collection of files that Visual Studio builds together to make a single program. You'll be creating C# source files, which carry the extension `.CS`. Project files use the extension `.CSPROJ`.

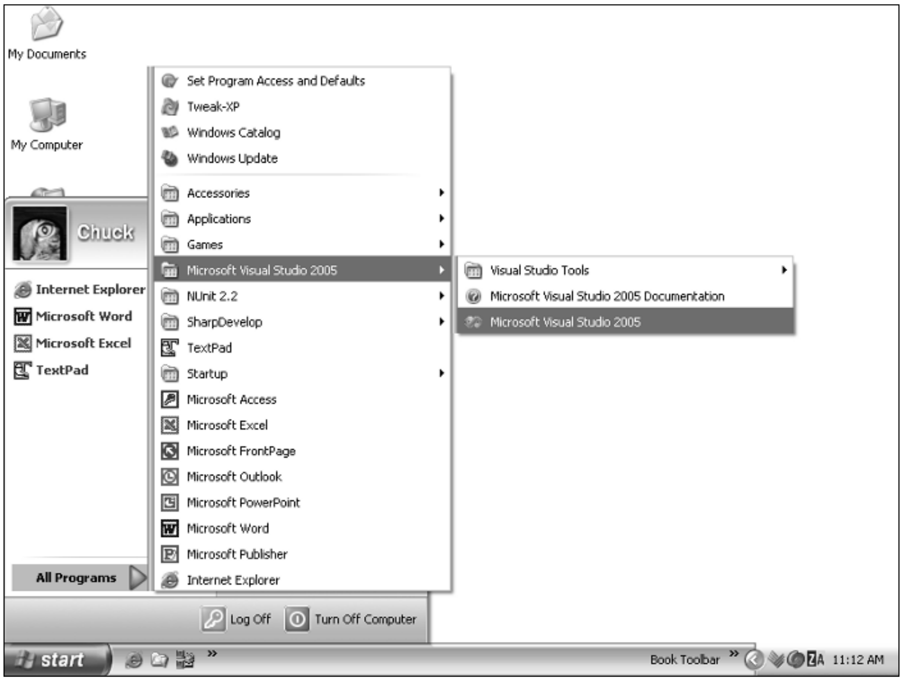


Figure 1-1: What a tangled web we weave when a C# program we do conceive.

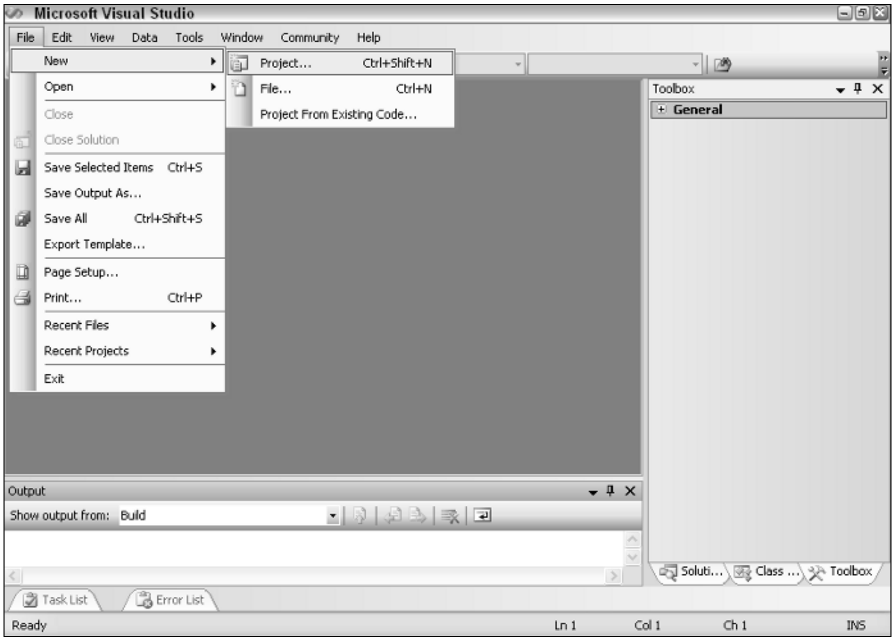


Figure 1-2: Creating a new project starts you down the road to a better Windows application.

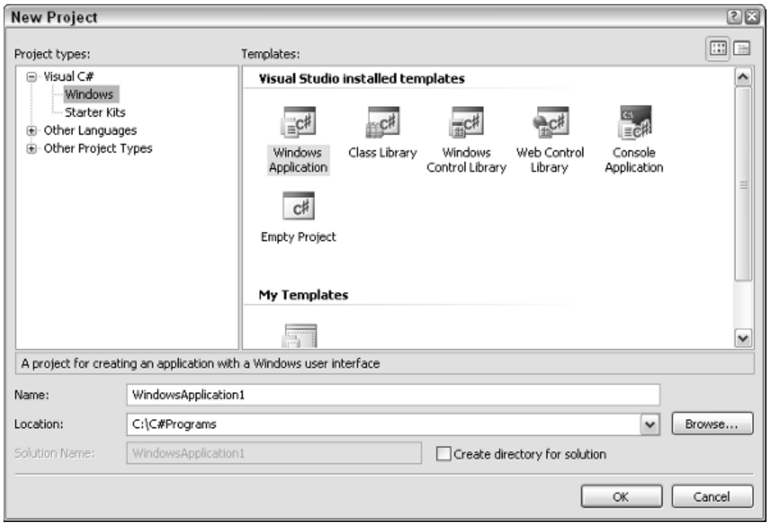


Figure 1-3: The Visual Studio Application Wizard is just waiting to create a new Windows program for you.

3. **Under Project Types, select Visual C#, and under that, click Windows. Under Templates, click Windows Application.**

   If you don't see the correct template icon right away, don't panic — you may need to scroll around in the Templates pane a bit.

   Don't click OK, yet.

4. **In the Name text box, enter a name for your project, or use the default name.**

   The Application Wizard will create a folder in which it stores various files, including the project's initial C# source file. The Application Wizard uses the name you enter in the Name text box as the name of that folder. The initial default name is `WindowsApplication1`. If you've been here before, the default name may be `WindowsApplication2`, `WindowsApplication3`, and so on.

   For this example, you can use the default name and the default location for this new folder: `My Documents\Visual Studio Projects\WindowsApplication1`. I put my real code there too, but for this book, I've changed the default location to a shorter file path. To change the default location, choose Tools➪Options➪Projects and Solutions➪General. Select the new location — `C:\C#Programs` for this book — in the Visual Studio Projects Location box, and click OK. (You can create the new directory in the Project Location dialog box at the same time. Click the folder icon with a small sunburst at the top of the dialog box. The directory may already exist if you've installed the example programs from the CD.)

5. **Click OK.**

   The Application Wizard makes the disk light blink for a few seconds before opening a blank *Form1* in the middle of the display.

Building and running your first Windows Forms program

After the Application Wizard loads the template program, Visual Studio opens the program in Design mode. You should convert this empty C# source program into a Windows Application, just to make sure that the template the Application Wizard generated doesn't have any errors.

REMEMBER

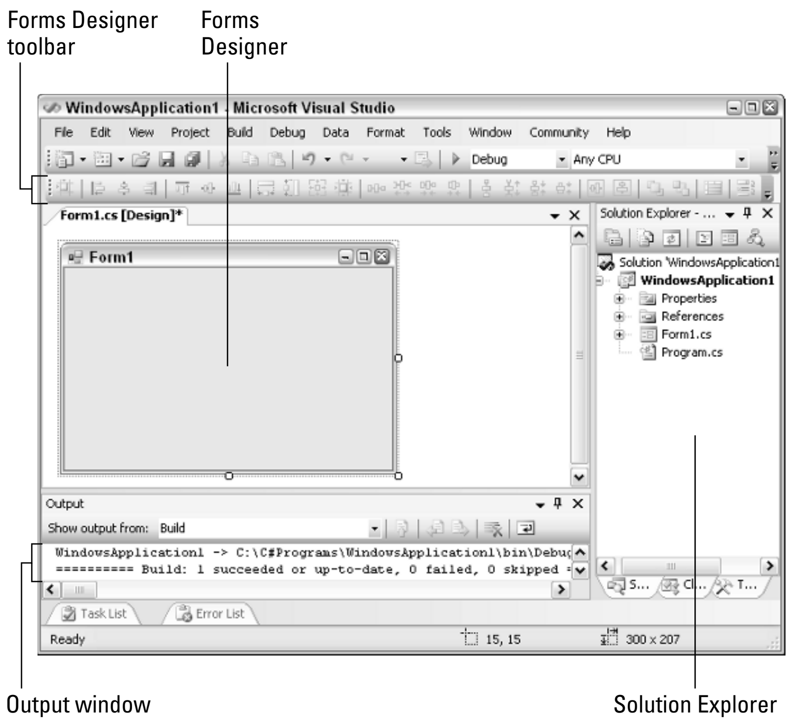
The act of converting a C# source file into a living, breathing Windows Application is called *building* (or *compiling*). If your source file has any errors, Visual C# will find them during the build process.

To build and run your first Windows Forms program, follow these steps:

1. **Choose Build➪Build *projectname* (where *projectname* is a name like WindowsApplication1 or MyProject).**

   The Output window may open. If not, you can open it before you build if you like. Choose View➪Other Windows➪Output. Then Build. In the Output window, a set of messages scrolls by. The last message in the Output window should be `Build: 1 succeeded, 0 failed,0 skipped` (or something very close to that). This is the computer equivalent of “No runs, no hits, no errors.” If you don’t bother with the Output window, you should see `Build succeeded` or `Build failed` in the status bar just above the Start menu.

Figure 1-4 shows what Visual Studio looks like after building the default Windows program, complete with Output window. Don’t sweat the positions of the windows. You can move them around as needed. The important parts are the Forms Designer window and the Output window. The designer window’s tab is labeled “Form1.cs [Design].”



```
WindowsApplication1 -> C:\C#Programs\WindowsApplication1\bin\Debug
========== Build: 1 succeeded or up-to-date, 0 failed, 0 skipped =
```

Figure 1-4: The initial Windows template program isn’t very exciting.

2. **You can now execute this program by choosing Debug➪Start Without Debugging.**

   The program starts and opens a window that looks just like the one in the Forms Designer window, as shown in Figure 1-5.

   In C# terms, this window is called a *form*. A form has a border and a title bar across the top with the little Minimize, Maximize, and Close buttons.

Figure 1-5: The template Windows application works, but it won't convince your spouse that Visual Studio 2005 is worth the price.

3. **Click the little Close button in the upper-right corner of the frame to terminate the program.**

   See! C# programming isn't so hard.

As much as anything, this initial program is a test of your installation. If you've gotten this far, your Visual Studio is in good shape and ready for the programs throughout the rest of this book.

TIP

Go ahead and update your resume to note that you are officially a Windows applications programmer. Well, maybe an application (as in one) programmer, so far.

Painting pretty pictures

The default Windows program isn't very exciting, but you can jazz it up a little bit. Return to Visual Studio and select the window with the tab `Form1.cs [Design]` (refer to Figure 1-4). This is the Forms Designer window.

The Forms Designer is a powerful feature that enables you to "paint" your program into the form. When you're done, click Build, and the Forms Designer creates the C# code necessary to make a C# application with a pretty frame just like the one you painted.

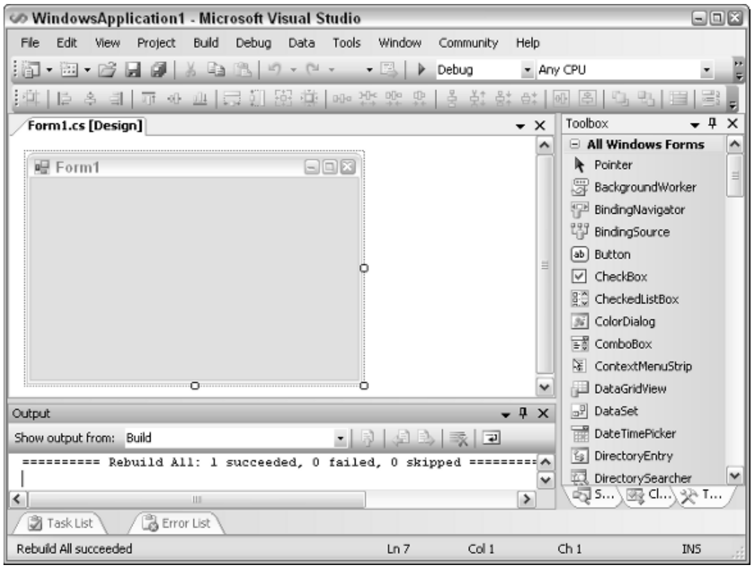
In this section, I introduce several new Forms Designer features that simplify your Windows Forms programming. You find out how to build an application with two text boxes and a button. The user can type into one of the text boxes (the one labeled Source) but not in the other (which is labeled Target). When the user clicks a button labeled Copy, the program copies the text from the Source text box into the Target text box. That's it.

Putting some controls in place

The labeled windows that make up the Visual Studio user interface are called *document windows* and *control windows.* Document windows are for creating and editing documents, such as the C# source files that make up a C# program. Control windows like the Solution Explorer shown in Figure 1-4 are for managing things in Visual Studio while you program. For much more about Visual Studio's windows, menus, and other features, read the first half of Bonus Chapter 4 on the CD that accompanies this book.

All those little doodads like buttons and text boxes are known as *controls.* (You also may hear the term *widget.*) As a Windows programmer, you use these tools to build the graphical user interface (GUI), usually the most difficult part of a Windows program. In the Forms Designer, these tools live in a control window known as the Toolbox.

If your Toolbox isn't open, choose View⇨Toolbox. Figure 1-6 shows Visual Studio with the Toolbox open on the right side of the screen.



```
========== Rebuild All: 1 succeeded, 0 failed, 0 skipped ==========
```

Rebuild All succeeded

Figure 1-6: The Visual Studio Toolbox is chock-full of interesting controls.

Don't worry if your windows are not in the same places as in Figure 1-6. For example, your Toolbox may be on the left side of the screen, on the right, or in the middle. You can move any of the views anywhere on the desktop, if you want. Bonus Chapter 4 on the CD explains how.

The Toolbox has various sections, including Data, Components, and Windows Forms. These sections, commonly known as tabs, simply organize the controls so you're not overwhelmed by them all. The Toolbox comes loaded with many controls, and you can make up your own.

Click the plus sign next to Common Controls (or the one labeled All Windows Forms) to reveal the options below it, as shown in Figure 1-6. You use these controls to jazz up a form. The scroll bar on the right enables you to scroll up and down within the controls listed in the Toolbox.

You add a control to a form by dragging the control and dropping it where you want. Follow these steps to use the Toolbox to create two text boxes and a button:

1. **Grab the Textbox control, drag it over to the form labeled `Form1`, and release the mouse button.**

   You might have to scroll the Toolbox. After you drag the control, a text box appears in the form. If the text box contains text (it may not), it says `textBox1`. This is the name the Forms Designer assigned to that particular control. (In addition to its Name property, a control has a Text property that needn't match the Name.) You can resize the text box by clicking and dragging its corners.

   REMEMBER

   You can only make the text box wider. You can't make it taller because by default these are single-line text boxes. The little right-pointing arrow on the text box — called a *smart tag* — lets you change that, but ignore it until you read Bonus Chapter 4 on the CD.

2. **Grab the Textbox control again and drop it underneath the first text box.**

   Notice that thin blue alignment guides — called *snaplines* — appear to help you align the second text box with other controls. That's a cool new feature.

3. **Now grab the Button control and drop it below the two text boxes.**

   A button now appears below the two text boxes.

4. **Resize the form and use the alignment guides as you move everything around until the form looks pretty.**

   Figure 1-7 shows the form. Yours may look a little different.

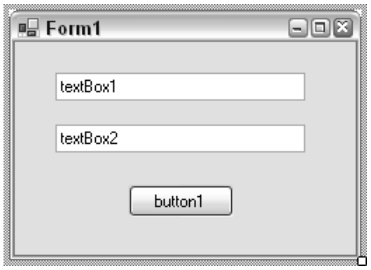


Figure 1-7: The initial layout of the form looks like this.

Controlling the properties

The most glaring problem with the application now is that button label. `button1` is not very descriptive. You need to fix that first.

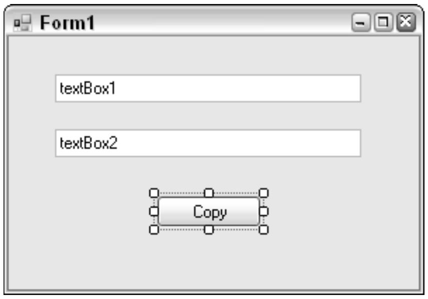
Each control has a set of properties that determine the control's appearance and the way it works. You access these properties through the Properties window. Follow these steps to change the properties of different controls:

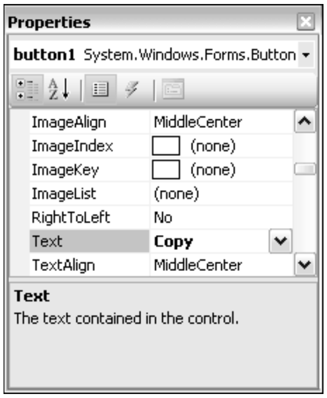
1. **Select the button by clicking it.**
2. **Enable the Properties window by choosing View➪Properties Window.**

   The button control has several sets of properties: the appearance set listed at the top, the behavior properties down below, and several others. You need to change the Text property, which is under Appearance. (To see the properties listed alphabetically rather than in categories, click the icon at the top of the window with AZ on it.)

3. **In the Properties view, select the box in the right-hand column next to the Text property. Type in** Copy **and then press Enter.**

   Figure 1-8 shows these settings in the Properties view and the resulting form. The button is now labeled Copy.




Properties

button1 System.Windows.Forms.Button

ImageAlign	MiddleCenter
ImageIndex	(none)
ImageKey	(none)
ImageList	(none)
RightToLeft	No
Text	**Copy**
TextAlign	MiddleCenter

Text
The text contained in the control.

Figure 1-8: The Properties view gives you control over your controls.

4. **Change the initial contents of the Textbox controls. Select the upper text box and repeat Step 3, typing the text** User types in here. **Do the same for the lower text box, typing the text** Program copies text into here**.**

   Doing this lets the user know what to do when the program starts. Nothing baffles users more than a confusing dialog box.

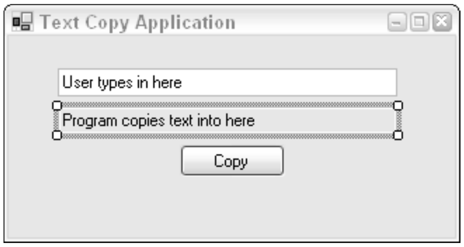
5. **Similarly, changing the Text property of the Form changes the text in the title bar. Click somewhere in the Form, type in the new name in the Text property, and then press Enter.**

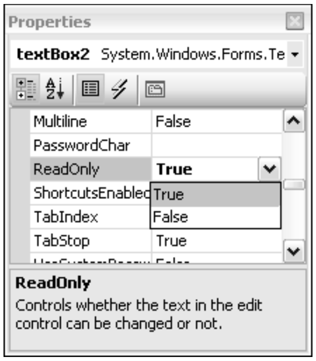
   I set the title bar to “Text Copy Application.”

6. **While you’re changing Form properties, click the AcceptButton property (under Misc in the Properties window). Click the space to the right of AcceptButton to specify which button responds when the user presses the Enter key. In this case, select `button1`.**

   “Copy” is the text on this button, but the name is still `button1`. You could change that too, if you like. It’s the Form’s Name property — a form property, not a button property.

7. **Select the lower text box and scroll through the Behavior properties until you get to one called ReadOnly. Set that to True by clicking it and selecting from the drop-down list, as shown in Figure 1-9.**





Multiline	False
PasswordChar	
ReadOnly	True
ShortcutsEnabled	True
TabIndex	False
TabStop	True

ReadOnly
Controls whether the text in the edit control can be changed or not.

Figure 1-9: Setting the text box to read only keeps users from editing the field when the program is executing.

8. **Click the Save button in the Visual Studio toolbar to save your work.**

   While you work, click the Save button every once in awhile just to make sure you don’t lose too much if your dog trips over the computer’s power cord. Unsaved files show an asterisk in the tab at the top of the Forms Designer window.

Building the application

Choose Build➪Build *WindowsApplication1* to rebuild the application. This step builds a new Windows Application with the Form you’ve just created. In the Output window you should see a `1 succeeded, 0 failed, 0 skipped` message.

Now execute the program by choosing Debug➪Start Without Debugging. The resulting program opens a form that looks like the one you've been editing, as shown in Figure 1-10. You can type into the upper text box, but you can't type into the lower text box (unless you forgot to change the ReadOnly property).

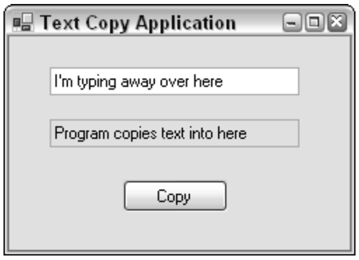


Figure 1-10: The program window looks like the Form you just built.

Make it do something, Daddy

The program looks right but it doesn't do anything. If you click the Copy button, nothing happens. So far, you've only set the Appearance properties — the properties that manage the appearance of the controls. Now, follow these steps to put the smarts into the Copy button to actually copy the text from the source text box to the target:

1. **In the Forms Designer, select the Copy button again.**
2. **In the Properties window, click the little lightning bolt icon above the list of properties to open a new set of properties.**

   These are called the control's *events.* They manage what a control does while the program executes.

   You need to set the Click event. This determines what the button does when the user clicks it. That makes sense.
3. **Double-click the Click event and watch all heck break loose.**

   The Design view is one of two different ways of looking at your application. The other is the Code view, which shows the C# source code that the Forms Designer has been building for you behind the scenes. Visual Studio knows that you need to enter some C# code to make the program transfer the text.

TIP

Instead of the lightning bolt, you can simply double-click the button itself on the Forms Designer.

When you set the Click event, Visual Studio switches the display to the Code view and creates a new *method.* Visual Studio gives this method the descriptive name `button1_Click()`. When the user clicks the Copy button, this method will perform the actual transfer of text from `textBox1`, the source, to `textBox2`, the target.

Don't worry too much about what a method is. I describe methods in Chapter 8. Just go with the flow for now.

This method simply copies the Text property from `textBox1` to `textBox2`, right at the blinking insertion point.

4. **Because `button1` is now labeled "Copy," rename the method with the Refactor menu. Double-click the name `button1_Click` in the Code window. Choose Refactor➪Rename. In the New Name box, type** CopyClick**. Press Enter twice (but take a look at the dialog boxes).**

   It's good for a control's text to reflect its purpose clearly.

   New in Visual Studio 2005, the Refactor menu is the safest way to make certain changes to the code. For instance, just manually changing the name `button1_Click` for the method would miss another reference to the method elsewhere in the code that the Forms Designer has generated on your behalf.

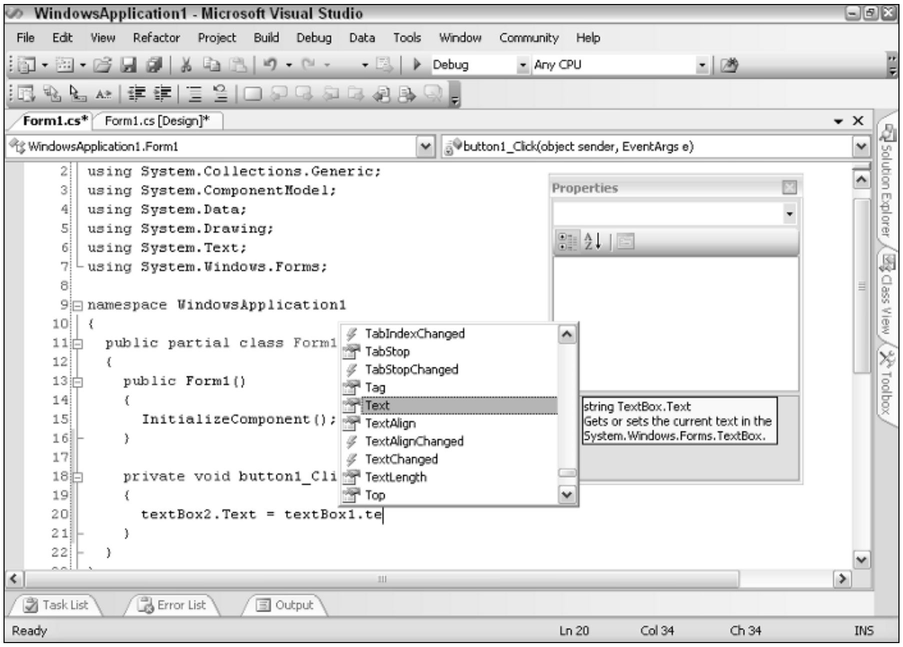
   The second dialog box for the Rename refactoring shows things that will change: the method and any references to it in comments, text strings, or other places in the code. You can deselect items in the upper pane to prevent them from changing. The lower Preview Code Changes pane lets you see what will actually change. Use the Refactor menu to save yourself lots of error-prone work.

5. **Add the following line of code to the `CopyClick()` method:**

   ```
   textBox2.Text = textBox1.Text;
   ```

   Notice how C# tries to help you out as you type. Figure 1-11 shows the display as you type the last part of the preceding line. The drop-down list of the properties for a text box helps to jog your memory about which properties are available and how they're used. This auto-complete feature is a great help during programming. (If auto-complete doesn't pop up, press Ctrl-Space to display it.)

6. **Choose Build➪Build *WindowsApplication1* to add the new click method into the program.**



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

namespace WindowsApplication1
{
  public partial class Form1
  {
    public Form1()
    {
      InitializeComponent();
    }

    private void button1_Cli
    {
      textBox2.Text = textBox1.te
    }
  }
```

string TextBox.Text
Gets or sets the current text in the System.Windows.Forms.TextBox.

Figure 1-11: The auto-complete feature displays the property names as you type.

Trying out the final product

Choose Debug➪Start Without Debugging to execute the program one last time. Type some text in the source text box and then click the Copy button. The text is magically copied over to the target text box, as shown in Figure 1-12. Gleefully repeat the process, typing whatever you want and copying away until you get tired of it.

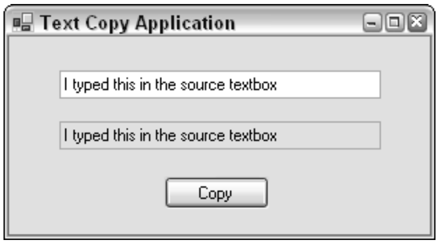

Text Copy Application
I typed this in the source textbox
I typed this in the source textbox
Copy

Figure 1-12: It works!

Looking back on the creation process, you may be struck by how picture-oriented it all is. Grab controls, drop them around on the frame, set properties, and that's about it. You only had to write one line of C# code.

You could argue that the program doesn't do much, but I would disagree. Look back at some of the earlier Windows programming books in the days before App Wizards, and you'll see how many hours of coding even a simple application like this would have taken.

Visual Basic 6.0 programmers, beware!

To those Visual Basic 6.0 programmers among you, this probably seems mundane. In fact, the Forms Designer works a lot like the one in later versions of Visual Basic. However, the .NET Forms Designer, which Visual C# uses, is much more powerful than its Visual Basic 6.0 counterpart. .NET and C# (and Visual Basic .NET, for that matter) use the .NET library of routines, which is more powerful, extensive, and consistent than the old Visual Basic library. And .NET supports developing distributed programs for the network as well as programs using multiple languages, which Visual Basic did not. But the chief improvement in the Forms Designer used by C# and Visual Basic .NET over the Visual Basic 6.0 predecessor is that all the code it generates for you is just code, which you can easily modify. In Visual Basic 6.0, you were stuck with what the designer gave you.