CHAPTER 1

INTRODUCING .NET

Welcome to .NET! I might as well have said, “Welcome to the Solar System,” because like the Solar System, .NET is huge. And it’s complex. And it’s filled with black holes and other things that don’t always make sense. Yet it (.NET, not the universe) turns out to be a fantastic system in which to develop software applications.

The .NET Framework was not developed in a vacuum (unlike the universe); Microsoft designed it and its related development languages—especially C# and Visual Basic—to address various issues that plagued Windows software developers and users. To fully understand why .NET was necessary, we need to take a short trip down computer memory lane.

Before .NET

Practical general-purpose computers have been around since the mid-twentieth century. However, they were inaccessible to most people because (a) they cost millions of dollars, (b) they consumed gobs of electricity, (c) maintenance and programming could only be done by highly-trained specialists, and (d) they tended to clash with the living room furniture.

Fast forward about 30 years. IBM comes out with the “personal” computer. These “desktop” computers represented a great advance in technology, but only a minority of people ever used them. They continued to be expensive (thousands of dollars), and maintenance and programming still required significant investments in training. IBM PCs also looked hideous around the living room furniture.

Then came the Apple Macintosh. With its sleek design and its user-friendly functionality, it introduced the joy of computing to the masses. And while programming it was not always straightforward, it did give nice results. It’s no wonder that Bill Gates decided to copy—oops, I mean improve upon—its functionality.

8009_ch01.qxd 9/18/06 11:24 AM Page 2

Microsoft Windows 1.0 brought a greater level of usability to the IBM/Intel computing platform. But it wasn't a free ride for programmers. MS-DOS development was hard enough without the addition of the "message pumps" and the hundreds of Application Programming Interface (API) calls needed by Windows programs. Visual Basic 1.0, introduced in 1991, greatly simplified the development process, but with the advent of 32-bit systems, ActiveX and COM components, and the Web, even VB programmers soon felt overwhelmed.

Throughout the 1990s, the situation only seemed to worsen. Microsoft saw increased competition in the form of the Java language and the Linux operating system. Hackers were exploiting buffer overruns and other security issues present in the Windows platform. Users experienced a myriad of computer problems stemming from conflicting standards, competing data integration technologies, registry bloat, and "DLL hell." In frustration, an Excel user's group set fire to the entire Microsoft campus in Redmond.

Well, it didn't get that bad. But Microsoft did see that it needed to address the overall software development and usability issues on its beloved Windows platform. Its solution came in the form of the .NET Framework.

Back to Introducing .NET

When Microsoft announced its plans for .NET, it surprised many developers, especially Visual Basic developers, who saw it as a giant step backward for "Rapid Application Development." But the release of the .NET Framework version 1.0 in 2002 did bring many needed benefits.

- *.NET introduced a unified programming environment.* All .NET-enabled languages compile to "Microsoft Intermediate Language" before being assembled into platform-specific machine code. Visual Basic and C# are language wrappers around this common .NET "language." Since all .NET-enabled compilers speak the same underlying language, they no longer suffer from the many data and language conflicts inherent in other component-based systems such as COM. The .NET version of Visual Studio also unified the standard user interface that lets programmers craft source code.

8009_ch01.qxd 9/18/06 11:24 AM Page 3

- *.NET committed developers to object-oriented technologies.* Not only does .NET fully embrace the object-oriented programming paradigm, *everything* in .NET is contained in an object: all data values, all source code blocks, the plumbing for all user-initiated events. Everything appears in the context of an object.
- *.NET simplified Windows programming.* Programming in Visual Basic before .NET was easy enough, until it came time to interact with one of the Application Programming Interface (API) libraries, something that happened a lot in professional programming. With .NET, most of these APIs are replaced with a hierarchy of objects providing access to many commonly needed Windows features. Since the hierarchy is extensible, other vendors can add new functionality without disrupting the existing framework.
- *.NET enhanced security.* Users and administrators can now establish security rules for different .NET features to limit malicious programs from doing their damage. .NET's "managed" environment also resolved buffer overrun issues and memory leaks through features such as strong data typing and garbage collection.
- *.NET enhanced developer productivity through standards.* The .NET Framework is built upon and uses many new and existing standards, such as XML and SOAP. This enhances data interchange not only on the Windows platform, but in interactions with other platforms and systems.
- *.NET enhanced Web-based development.* Until .NET, a lot of Web-based development was done using scripting languages. .NET brings the power of compiled, desktop development to the Internet.
- *.NET simplified the deployment of applications.* If .NET is installed on a system, releasing a program is as simple as copying its EXE file to the target system (although an install program is much more user-friendly). Features such as side-by-side deployment, ClickOnce deployment (new in 2005), and an end to file version conflicts and "DLL hell" (the presence of multiple versions of the same DLL on a system, or the inability to remove a version of a DLL) make desktop and Web-based deployments a snap.

If you didn't understand some of the terms used in this section, that's all right. You will encounter them again, with explanations, in later chapters.

8009_ch01.qxd 9/18/06 11:24 AM Page 4

The .NET Object

To fully understand software development in .NET, you must understand what an **object** is. (If you are familiar with object-oriented programming—OOP—then you can probably skip down to the next section, although you will miss some really great content.) While some of this section's information will also appear in Chapter 8, "Classes and Inheritance," it is so important to the discussion of .NET that a portion appears here as well.

Objects and Data

From a programming standpoint, a computer performs four basic tasks:

1. It stores *data* in the computer's memory area.
2. It supports processing of this *data* through basic operations, including addition and subtraction, Boolean algebra, and text string manipulation.
3. It allows the user to interact with the *data* stored in memory.
4. It provides a way to bring the *data* in and out of memory, through input and output devices such as keyboards and printers, and through long-term storage media, such as hard drives.

The core of these four activities is **data**. Computers exist to manipulate data. Operating systems provide the basic foundation for these activities, but it is software applications that make these features—the ability to manipulate data—real and meaningful to the user. High-level programming languages are the primary tools used to develop these applications, each of which uses some general methods to make data manipulation features available to the programmer. Back in the good old days of assembly language development, if you knew the memory address of a piece of data, you could access and manipulate it directly. In early flavors of BASIC and in most other "procedural" languages, data was accessed through **variables**.

As languages grew in complexity and purpose, so did their view of data. In the LISP (short for "List Processing" or "Lots of Irritating Silly Parentheses") language, any data value exists within a larger *list* or *set* of data. But in .NET languages, data is viewed through the *object*.

8009_ch01.qxd 9/18/06 11:24 AM Page 5

Objects are collections of data values and associated source code. While in older BASIC dialects, each data element was more or less independent through its named variable, related data values in OOP languages can be grouped into objects. Objects often include source code designed to manipulate the data values of that object.

Objects generally represent some *thing*, often a thing that has a real-world counterpart, whether physical or conceptual. For instance, your code may include a *House* object that has data **fields** or **properties** for the address, the exterior paint color, and the number of people living in the house. Associated source code could manage that data; a *Paint* **method** could alter the color value used for the exterior paint.

The data and code elements within an object are called **members**. Some members are hidden inside the object and can only be accessed by the object's source code. Other members are more public; any code in your application can use them, not just that subset of application code found inside the object. Consider a television as an object (see Figure 1-1).

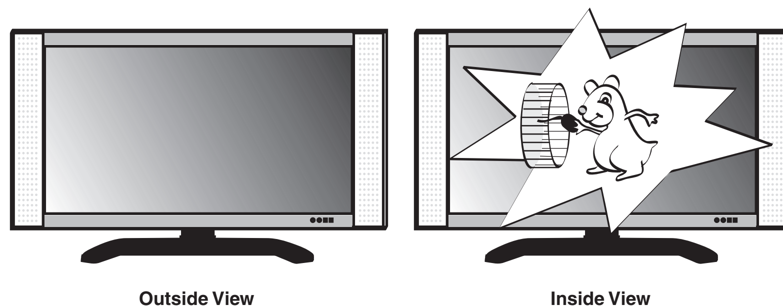

Figure 1-1 A TV: It's an object, not just objectionable.

The public members of a TV are generally easy to use: the power button, channel selector, volume control, and so on. They are the conduits through which the user controls the data values of the TV (its video and audio output). There are also hidden members inside of the TV; you could use these members to impact the picture and sound quality, although this would be a bad idea for most users. You don't want me messing with the internal members of your TV set, trust me. In the same way, an object doesn't want code outside of the object to mess with its internal members

8009_ch01.qxd 9/18/06 11:24 AM Page 6

except through the public members. I don't care how a TV works internally, as long as I can get pictures and sound out of it by using the controls that are exposed (power, channel, volume).

Objects and Interfaces

The public members of an object represent its **interface**. If code outside of the object wants to manipulate the data belonging to that object, it uses the members of the interface. It doesn't have to figure out the hidden members or how they work, and that's good. It's especially good if those internal members ever change for any reason, which happens more often then you think. Consider how the internals of TVs have changed just in the last 30 years. Here's a drawing of the TV my family had when I was a kid. Compare it to modern flat-screen TVs available today (see Figure 1-2).

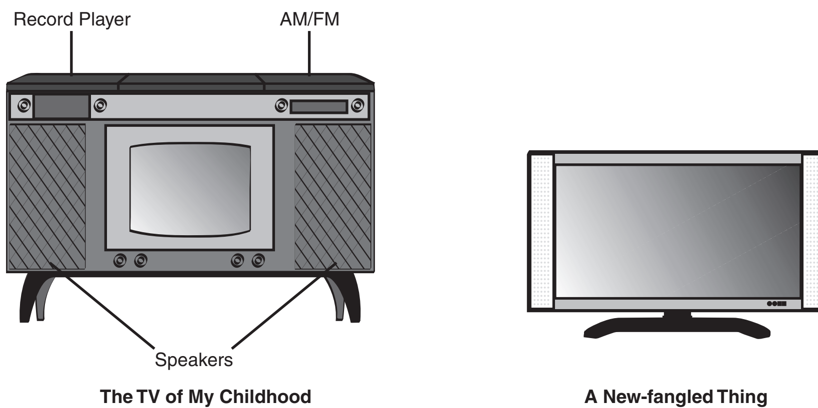

Figure 1-2 Are those really TVs?

My family's TV was cool. It had an AM/FM stereophonic hi-fi radio, a turntable that could play 33-1/3, 45, *and* 78 RPM records, and a large 19-inch display with vivid black and white crystal clear display. You could hide two kids behind it when playing hide and seek. And my friend who had the same model said that you could draw these really cool permanent lines on the screen with a magnet. Who cares that the speaker panels looked like vertical shag carpet? Who cares that the unit took up thirty percent of the floor space in the room? Who cares that you could cook sausages on top of

8009_ch01.qxd 9/18/06 11:24 AM Page 7

it from the heat generated by the vacuum tubes? It was more than a TV; it was an *entertainment center*.

Now compare it to the wimpy little flat screen job on its right. If you look closely, you find that the interface to the TV hasn't really changed much in three decades. There are still controls for power, volume, and channel selection (although Horizontal Hold and Vertical Hold are gone, sniff). What has changed is the internal configuration. Gone are the humming vacuum tubes, all replaced with efficient transistors and solid-state components. But it doesn't really make much difference to the TV viewer, since the public interface remains the same.

Objects in OOP development work in the same way. As long as the public interface remains the same, the object's actual code and internal data storage system—also known as the object's **implementation**—can change with no impact to the overall application.

Objects and Instances

The interface and implementation of an object really only represent its design; these are the parts created through the source code. They exist even before the program is compiled and installed on the user's computer. In fact, at this level, objects really aren't even known as objects. In most languages (including Visual Basic), the word **class** indicates the implementation of an object's interface.

Once your application is installed on a computer and starts up, the code creates **instances** of the class to store actual data in memory. These instances are the true objects of OOP development. Depending on how your code is written, a single class implementation might create just one or hundreds of objects in memory at the same time.

In .NET, all of your code and data values appear inside of objects. Pretty much everything you see in a running program is an object: Windows forms are objects; a list box control on that form is an object; and a single item in that list box is an object.

The Parts of the .NET Framework

So now you know all about objects, and you are probably thinking it's time to toss this book into the pile and start programming. But there are a few more parts of the .NET Framework still to discuss. These parts show up *ad*

8009_ch01.qxd 9/18/06 11:24 AM Page 8

nauseum in the .NET documentation, and they each have a three-letter acronym (TLA), or thereabouts.

The Common Language Runtime

At the center of the .NET Framework is the **Common Language Runtime** (CLR), so named not because it is *common* or ordinary, but because all .NET-enabled languages share it in *common*. Everything you do in a .NET program is *managed* by the CLR. When you create a variable, thank the CLR and its data *management* system. When you say goodbye to a piece of data, thank the CLR and how it *manages* the release of data through its garbage collection system. Did you notice how the word "manage" keeps showing up in those sentences? My editor sure did. But "manage" is the *mot juste*, since that is what the CLR does. In fact, software written for the .NET Framework is called **managed code**. Any code that falls outside of the CLR's control, including COM (ActiveX) components used by your .NET application, is known as **unmanaged code**.

The CLR is a lot like Los Angeles International Airport. If you have ever been to LAX, you know that there is a whole lot of activity going on. Airplanes arrive and depart each minute. Cars by the thousands enter and leave the two-level roadway and the central parking structures. People and pickpockets move constantly between the eight main terminals and the massive international terminal. There's a lot happening, but so much of it is managed. Planes cannot take off or land without approval from the control tower. Access points and gates manage the roadways and parking garages. Friendly, courteous security personnel manage the flow of passengers and pickpockets into and out of the secure areas of the terminals.

The control and management structures in place at LAX ensure an orderly and secure flow of people between their planes and the city of Los Angeles. The control and management structures of the CLR ensure an orderly and secure flow of data between .NET code and the rest of the computer or connected network.

You'd probably like to know the secret of how the CLR is able to process programs written in any .NET language, including Visual Basic, C#, and Fortran. So would Microsoft's competitors. Actually, they do know, because there is no secret. All .NET-enabled languages convert (that is, "compile") your source code into **Microsoft Intermediate Language** (or **MSIL**, pronounced "missile," and more commonly abbreviated as just **IL**).

8009_ch01.qxd 9/18/06 11:24 AM Page 9

For those of you familiar with assembly language, it looks a lot like that. For those of you not familiar with assembly language, it looks a lot like gibberish. For example, here is some Visual Basic source code for a **console application** (a non-Windows text-based program, like the old MS-DOS programs) that simply outputs "Hello, World!" from a code procedure called "Main."

```
Module Module1
   Sub Main()
      Console.WriteLine("Hello, World!")
   End Sub
End Module
```

That's the whole .NET program. When the Visual Basic compiler converts it to MSIL, the "Main" procedure looks like this (slightly modified to fit on this page).

```
.method public static void  Main() cil managed
{
  .entrypoint
  .custom instance void [mscorlib]System.
     STAThreadAttribute::.ctor() = ( 01 00 00 00 )
  // Code size       11 (0xb)
  .maxstack  8
  IL_0000:  ldstr    "Hello, World!"
  IL_0005:  call
     void [mscorlib]System.Console::WriteLine(string)
  IL_000a:  ret
} // end of method Module1::Main
```

Yes, it is gibberish. But that's OK, because it fulfills the International Computer Book Association's requirement that every Chapter 1 include a "Hello, World" code sample. Also, the CLR understands it, and that's what really counts in .NET. As long as you can get your code into IL, .NET will process it. The Visual Basic compiler just happens to generate IL for you. Other .NET language compilers, including C#, target IL as well. You can even write your own IL code, but you're probably reading the wrong book for that. Just to put your mind at ease, this will be the last bit of IL you will see in this book.

8009_ch01.qxd 9/18/06 11:24 AM Page 10

The Common Language Specification

Languages that claim to support .NET cannot just say so for any old reason. They truly have to be compatible with .NET and its workings. This is done through the **Common Language Specification** (CLS). The CLS defines a minimum set of features that a language must implement before it is considered to be .NET-compliant, or more accurately, CLS-compliant.

A language can go way beyond that minimum if it wants, and .NET includes many additional features upon which language-specific features may be built. A language that only implements the minimum CLS-specified features may not be able to fully interact with components from languages that exceed the minimum specification. Visual Basic is, of course, CLS-compliant, and in fact goes way beyond that minimum.

The Common Type System

Since the CLR is controlling your source code anyway, Microsoft thought it would be good to have it control the source code's data as well. The .NET Framework does this through its **Common Type System** (CTS), which defines all of the core data types and data mechanisms used in .NET programs. This includes all numeric, string, and Boolean value types. It also defines the **object**, the core data storage unit in .NET.

The CTS divides all data objects into two buckets. The first bucket, called **values types**, stores actual data right in the bucket. If you have a 32-bit integer value, it gets put right in the value type bucket, ready for your immediate use. The other bucket contains **reference types**. When you look in this bucket, you see a map that tells you where to find the actual data somewhere else in the computer's memory. It seems like value types are easier to use, and they are, but they come with a few restrictions not imposed on reference types.

Programs and components written using the CTS standard can exchange data with each other without any hindrances or limitations. (There are a few .NET data types that fall outside of the "core" CTS types, but you only need to avoid them when you want to specifically interact with components that can only use the core CTS types.)

When you write your applications in Visual Basic, most of your code will appear in **classes**. Classes are reference types that include both data values and associated code. The data values included in a class are most often the core CTS data types, but they can also contain objects that you design elsewhere in your application. Visual Basic also includes

8009_ch01.qxd 9/18/06 11:24 AM Page 11

structures, the weaker yet quicker younger brother of classes. Structures implement value types, and also include both data and code.

Classes and structures are just two of the data/code types available in Visual Basic. **Interfaces** are class and structure skeletons; they include design details of what should appear in a related class or structure, but don't include any actual implementation or working code. **Delegates** define a single procedure (but not its implementation), and are used to support **events**, those actions (initiated by the user, by the operating system, or by your code) that tell your code, "get to work now!" **Sea otters** are aquatic mammals that are curiously related to the weasel, and like to eat sea urchins. **Modules** are blocks of code and data, but unlike classes and structures, you can't create independent objects from them. **Enumerations** group a set of related integer values, generally for use as a list of choices.

In .NET parlance, all of these terms (class, structure, interface, delegate, module, and enumeration, but not sea otter) are known collectively as **types**. You probably already knew that .NET had some confusing elements in it; you wouldn't have bought a book about it if it was easy. But despite all of the complex technology, it is this simple word "type" that causes the most confusion. You will likely experience some angst throughout this book each time you read it. The problem: It's too general. Not only does it refer to these core elements of the Common Type System, but it is also used when talking about just the Visual Basic-specific value types (more often called the Visual Basic "data types"). The nickname for structures is "user-defined types," yet another confusing use of "type." Programmers who used Visual Basic before its .NET incarnation also remember "Type" as the language statement used to create user-defined types. Arrrgh! Microsoft should have used some word other than "types" for the world of classes, interfaces, enumerations, and so on. "Bananas" would have been a better choice since it is only sometimes used to discuss software. But "type" *is* the word, so you better get used to seeing it. I will try to include as much context as possible when using the word throughout this volume.

The members of a type usually consist of simple data fields and code procedures, but you can also include other types as members. That is, a class can include a **nested** class if it needs to. Only certain types support nesting (see Chapter 8 for details). I also talk about **access levels** in that chapter. Each member has an access level that says what code can use that member. There are five access levels, ranging from **Public** (anybody and their brother can use the member) to **Private** (you have to be inside the type to even know it's there).

8009_ch01.qxd 9/18/06 11:24 AM Page 12

Chapter 6, "Data and Data Types," discusses the .NET type system in greater detail, including the information you crave on classes, structures, and other bananas.

.NET Class Libraries

Computers are actually quite stupid. While I can count all the way to 17, a computer tops out at 1; it only knows the digits 0 and 1. The CPU includes a set of simple operators used to manipulate the digits 0 and 1, and a few more operators that compare 1's and 0's in complex ways. The computer's last great trick is its ability to move 0's and 1's into and out of memory, but whoop-dee-doo. Sure it does these things at nearly the speed of light, but can it calculate π to three million decimal places?

Well, actually it can. Computers don't know anything about the letters of the alphabet, and they really only can handle the digits 0 and 1, yet here I am using a computer to write an award-winning book. It is the ability to combine the simple one-bit data values and operators into increasingly complex *libraries* of functionality that make useful computers possible.[1]

The .NET Framework is built upon decades of increasingly complex functionality. When you install the .NET Framework, the CLR and its associated type system represent the core of the framework. By itself, the framework includes all of the basic functionality needed to let you add 2 and 2 together and correctly get 4. And as a business application developer, you spend a lot of time doing just that. But what if you want to do something more complex, something that you know some other programmer has already done, like sorting a list of names or drawing a colored circle on a form? To get that answer, go to the **class libraries**, the .NET Class Libraries. These libraries, installed with the Framework, include a lot of pre-written (increasingly complex) functionality that you don't have to write from scratch.

There are two class libraries in .NET: the **Base Class Library** (BCL) and the **Framework Class Library** (FCL). The BCL is smaller, and contains the most essential features that a program just couldn't do without. It includes only those classes that are an absolute must for supporting applications on the framework if Microsoft were, say, to port the framework to Linux.

---

1. If you want to read a truly fascinating book on how complex software and hardware operations are formed from the most basic uses of 0 and 1, read Charles Petzold's book *Code: The Hidden Language of Computer Hardware and Software* (Microsoft Press, 1999).

8009_ch01.qxd 9/18/06 11:24 AM Page 13

The FCL is larger, and includes everything else Microsoft thought you would want to have in your programs, but was not absolutely essential to have in the BCL. Don't even ask how many classes there are in the FCL; you don't want to know. I bet that Microsoft doesn't even really know the full number. I am convinced that those wacky pranksters at Microsoft have included "gag" classes in the FCL, but they are so deeply buried that few programmers ever encounter them.

With thousands (yes, thousands!) of classes, enumerations, interfaces, and other types included in the BCL and FCL, you would think that it would be hard to find just the class you need. But it's not that difficult, at least not overwhelmingly difficult. The .NET Framework includes a feature called **namespaces**. All types in .NET appear in a hierarchy—a tree-like structure—with just a few minimal entries at the root. Each **node** in the hierarchy is a namespace. You uniquely identify any class or other type in the libraries by naming all of the namespaces, from the root down to the local namespace that contains the class, separating each node with a period (.).

Unlike most hierarchies that have all branches starting from a single root node, the .NET namespace hierarchy has multiple root nodes. The largest root namespace is named **System**. It includes many classes, but it also includes several next-tier hierarchy nodes (namespaces). Since the framework includes features for both Windows-based and Web-based application development, there are namespaces that contain the Windows-specific and Web-specific development features. These namespaces appear just within the *System* namespace, and are called **Windows** and **Web**. All code related to on-screen Forms in the *Windows* namespaces appears in the **Forms** namespace, and within this namespace is the actual class that implements a form, named **Form**. Figure 1-3 presents an image of this namespace subset.

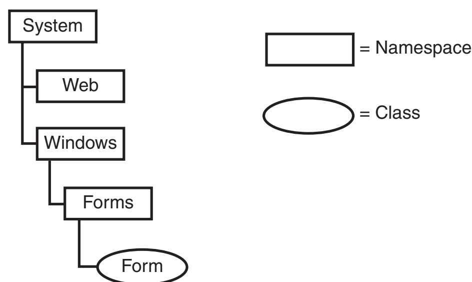

Figure 1-3 A hierarchy of namespaces and classes

8009_ch01.qxd 9/18/06 11:24 AM Page 14

In Visual Basic, you identify a class by qualifying it with all of its namespaces, starting from its root namespace. The *Form* class has the following fully-qualified name:

```
System.Windows.Forms.Form
```

All classes and types exist somewhere in the hierarchy, although not every class descends from *System*. Many of the supporting features specific to Visual Basic appear in the *Microsoft.VisualBasic* namespace, which has "Microsoft" as its root node instead of "System." When you create new projects in Visual Basic, the name of the project is, by default, a new top-level node in the hierarchy. If you create a new Windows application, the default "Form1" form has the following fully-qualified name:

```
WindowsApplication1.Form1
```

This new application's namespace is not just a second-class appendage hanging off of the *System* namespace. It is fully integrated into the full .NET namespace hierarchy; the *WindowsApplication1* namespace is a root node, just like the *System* and *Microsoft* root nodes. Visual Basic includes features that let you alter the default namespace for your application, or place one of the application's classes in a specific namespace. You can even place your application's classes in the *System* namespace branch. Changing *WindowsApplication1* to *System.MySuperApp* moves *Form1* to:

```
System.MySuperApp.Form1
```

If your application is actually a component or library destined for use in programs, your app's classes will appear in the namespace you specify when the other program loads your component into its application area. Your code will look like it is part of the Microsoft-supplied namespaces. Is that cool or what?

While you can add your classes to the *System* namespace, you will incur the wrath of other .NET programmers. The *System* namespace is supposed to be for "system" (read: Microsoft-supplied) features, and that's it. Also, there's a chance that two vendors might use the same namespace path. So, to avoid potential namespace conflicts *and* dirty looks from other programmers, you should name your application's classes as:

```
CompanyName.ApplicationName.ClassName
```

A single class or other type cannot be split across multiple namespaces, even within the same hierarchy branch. However, two classes or types may share a common name in different namespaces, even within the same branch.

All classes of the BCL and FCL appear intermingled throughout the entire namespace hierarchy. This means that you cannot necessarily tell whether a particular class is from the BCL or the FCL. Frankly, it doesn't really matter; your code won't care which library a class comes from, as long as it is available for use on the user's workstation.

Assemblies and Manifests

An **assembly** is a "unit of deployment" for the parts of a .NET application or library. In 99.9% of cases, an assembly is simply a .NET executable file (an "exe" file) or a .NET library of classes and other types (a "dll" file). It is possible to split an assembly between multiple files, but usually it is one file for one assembly.

What makes an ordinary *exe* or *dll* file an assembly is the presence of a **manifest**. For single-file assemblies, the manifest appears right in the file; it can also appear in a file of its own. The manifest is a chunk of data that lists important details about the assembly, including its name, version information, default culture, information on referencing external assemblies and types, and a list of all the files contained in the assembly. The CLR will not recognize an assembly without its manifest, so don't lose it.

Assemblies can include an optional **strong name**. This helps to ensure the integrity and authenticity of an assembly through a digital signature attached to the manifest. The strong name uses public key cryptography to guarantee that the assembly is unique and has not been tampered with. Visual Studio and the .NET Framework include tools that let you add a strong name to an assembly.

When you deploy your application, you will normally place all assembly files, configuration files, and any related files specific to your application in the application's install directory, just like in the old Jurassic days before .NET. Shared assemblies designed to be used by more than one application on a single machine can be stored in the **global assembly cache** (GAC). All assemblies placed in the GAC must have strong names. Some systems may only allow the system administrator to add assemblies to the GAC.

8009_ch01.qxd 9/18/06 11:24 AM Page 16

Metadata and Attributes

Assemblies are brought to you by the letter "m." In addition to *manifests* and type *members*, assemblies also contain **metadata**. The application code and data elements stored in an assembly parallel the code and data items found in the related Visual Basic source code; for each type and member in your source code, there is associated executable code in the deployed assembly. This makes sense, and is not much of a change from pre-.NET deployments. What is different is that the Visual Basic compiler now attaches additional information—metadata—to each type and member in the assembly. This metadata documents the name of the associated content, information about required data types, information on class inheritance for the element, and security permissions required before the element can be used by the user or other software.

Your Visual Basic source code can enhance the metadata for any element of your assembly through **attributes**. The metadata generated by an attribute is more than just some ID number. Attributes implement full .NET classes, with their own data values and associated logic. Any .NET code that knows how to process attributes can examine the attributes for a type or member and take action as needed. This includes Visual Studio, the Visual Basic compiler, and your own custom applications.

How's this for a mundane example: The .NET Framework includes an attribute named *ObsoleteAttribute*. This attribute lets you mark types or members of your assembly as obsolete or no longer supported. (Visual Studio uses this attribute to display a warning whenever you attempt to use an out-of-date BCL or FCL feature.) To use the attribute, add it to a member of your application using angle brackets.

```
Class MyClassWithOldMembers
   <ObsoleteAttribute> Sub DoSomeWork()
   End Sub
End Class
```

This code defines a single class (*MyClassWithOldMembers*) with a single member procedure (*DoSomeWork*), a procedure that clearly does some work. The procedure is tagged with the *ObsoleteAttribute* attribute. By custom, all attribute names end in the word "Attribute." You can leave off this portion of the word if you wish, as long as the resulting word does not conflict with any Visual Basic language keywords.

8009_ch01.qxd 9/18/06 11:24 AM Page 17


```
Class MyClassWithOldMembers
   <Obsolete> Sub DoSomeWork()
   End Sub
End Class
```

When you compile the class and store it in an assembly, the `<ObsoleteAttribute>` attribute is stored as part of *DoSomeWork*'s definition. You can now write a separate Visual Basic application that scans an assembly and outputs the name and status of every type and member it finds. When that analysis program encounters the obsolete member, it would detect *ObsoleteAttribute* in the metadata, and output the status:

```
DoSomeWork Procedure: Obsolete, don't use it!
```

Most attributes are designed with a specific purpose in mind. Some attributes instruct Visual Studio to display the members of a class in specific ways. You've probably already played with the form-editing features of Visual Studio to design a simple Windows desktop application. When you add a control (such as a button or a list box) to a form and select that control, Visual Studio lets you edit details of that control through the Properties panel area (see Figure 1-4).

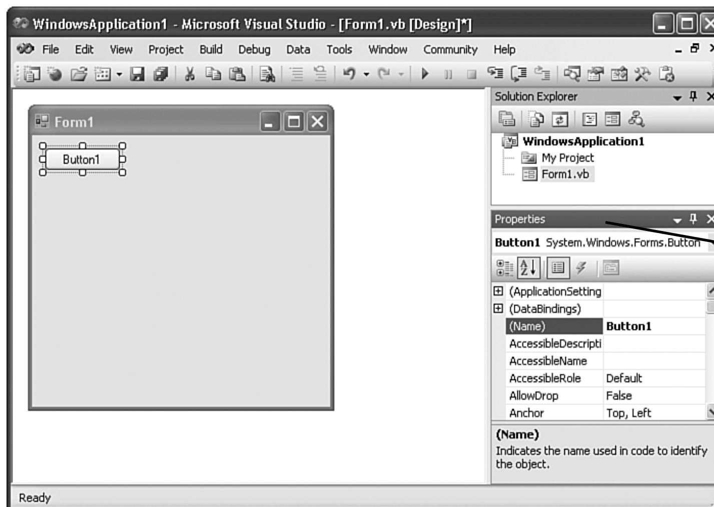

Figure 1-4 The Properties panel in Visual Studio

8009_ch01.qxd 9/18/06 11:24 AM Page 18

The Button control is implemented as a class, and many of its class members appear in the Properties panel, but not all of them. When the Button class was designed, attributes were added to its members that tell Visual Studio which members should appear in the Properties panel, and which should not. Visual Studio dutifully examines these attributes, and only displays the requested properties.

Versioning

Like you, my applications are perfect from their initial release, and I never have a reason to modify them or add additional features. But there are software development organizations—including one large company that, so as not to cause embarrassment, I will refer to only by its initial letter of "M"—that feel the need to "one-up" their competition by coming out with "improved" versions of their previously released software offerings. Let's say that "M" happened to have a popular word processor that includes version 1.0 of a spell-check component. "M" also happens to sell an email tool that depends specifically on version 1.0 of that same shared component. If, in a show of competitive machismo, "M" releases an update to the word processor *and* the spell-check component (now version 2.0), what happens to the email tool's spell-checking ability?

Not that this ever happens in real life. But if it did, the replacement of a vital shared component with a newer but somewhat incompatible version could cause real problems. A related problem is the deployment of multiple versions of a single component on the same workstation, all in different directories. Can any of them be safely deleted?

.NET solves these problems through **versioning**. All assemblies that use shared components identify exactly which versions of the shared components they require. While an application can be reconfigured to use a later version, it will only use the originally specified version of a shared component by default.

Multiple versions of a single shared component can be added to the Global Assembly Cache, a feature called **side-by-side deployment**. The CLR ensures that the right application links up with the right component. You can even run applications simultaneously that use different versions of the same component.

From Source Code to EXE

Now you know pretty much everything there is to know about .NET except for that pesky programming thing. Before delving into some actual code, let's take a little snack break and examine the lifetime of an application, from start to finish (see Figure 1-5).

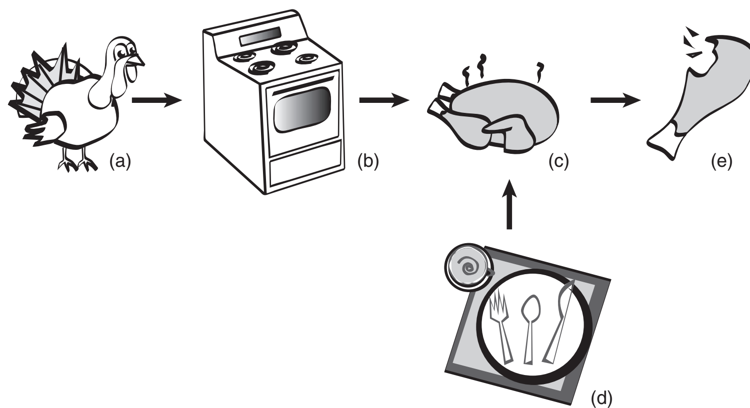

Figure 1-5 The real Visual Basic development process

So here's what happens, step by step:

1. You, as the programmer, are responsible for preparing the basic ingredients *(a)* of the application. For Visual Basic programs, this means creating one or more source code files with a ".vb" extension. Your ingredients may also include other support files, such as resource files (text and graphic files, often used for multi-language support).
2. Your application is cooked by the Visual Basic compiler *(b)*. The result is an assembly, complete with a manifest and metadata. The output is actually semi-compiled MSIL and includes ready-to-execute versions of the original source code's types and members, including all member and type names. All this content can be

8009_ch01.qxd 9/18/06 11:24 AM Page 20

"decompiled" (returned back to full MSIL, although not to full Visual Basic) using a tool named *ildasm.exe* (the Microsoft Intermediate Language Disassembler), which is included with the .NET Framework. Since you probably don't want just anyone disassembling your application and looking at the code, Microsoft (and other third parties) also supplies an **obfuscator**, which sufficiently scrambles the content of your code to make it just difficult enough to discourage prying eyes.

3. The assembly *(c)* is deployed to the user's workstation. There are a few different methods used to deploy the application, including (1) generating a standard Windows Installer setup package, (2) generating a **ClickOnce** deployment, which is new with version 2.0 of .NET, or (3) performing an **xcopy** install, which involves nothing more than copying the EXE assembly itself to the destination machine. No matter which deployment method you choose, the .NET runtime *(d)* must also be installed on the user's workstation.
4. The user eats—I mean runs—the program *(e)*. The CLR does a final **just-in-time** (JIT) compile of the MSIL assembly, to prepare it for use on the local platform. It then presents the application to the user, and manages all aspects of the application while it runs. The user experiences a level of joy and satisfaction rarely encountered when using other software applications.

As with the preparation of a Thanksgiving meal, the actual development process is somewhat more involved than just reading a paragraph (or a recipe book) about it. But it's not so difficult that it can't be put in a book, a book like this one.

What About Visual Studio and Visual Basic?

Wait a minute, what about Visual Studio? That last section didn't even mention it. And it didn't need to, since *you do not need to use Visual Studio to develop, compile, deploy, or run Visual Basic applications*. The entire .NET Framework—including the Visual Basic compiler—is available for free from Microsoft's web site; download it and use it to develop and deploy applications that are every bit as powerful and complex as, well, Visual Studio.

8009_ch01.qxd 9/18/06 11:24 AM Page 21

The July 1983 issue of *Datamation* magazine includes a letter from manly reader Ed Post entitled, "Real Programmers Don't Use Pascal."[2] I highly recommend that you read this article, as it will help you quickly separate the real programmers from the "quiche eaters." And when you do, run away as fast as you can from the real programmers. Oh sure, they can reconstruct your source code from the obfuscated .NET assembly, but they will be useless on a team project using Visual Studio.

A "real programmer" could code any .NET application using Notepad, and it would run. Actually, they would use *emacs* or *vi* instead of Notepad (since Windows does not include a keypunch interface), but the results would be the same. They would growl as you blissfully type away in Visual Studio's elegant, well-designed, and fully customizable and extensible user interface. They would gripe and bare their cheese-cracker-with-peanut-butter-encrusted teeth at you while you use the IntelliSense and AutoCompletion features built into the Visual Studio code editor. They would consume another slice of quiche-shaped cold pizza while you drag-and-drop both Windows and web-based user interfaces.

Yes, the real programmer could generate full applications with just a text (or hex) editor and a .NET compiler, but you would get the glory, since you would be done in a fraction of the time it would take the FORTRAN lover to eek out his code.

Visual Studio 2005

Since this is a book on Visual Basic development and not on Visual Studio usage, I won't be delving too much into Visual Studio's features or its user interface elements. It is a great application, and its tight integration with the .NET Framework makes it the best tool for developing applications with .NET. But as the real programmer would tell you, it is really just a glorified text editor. Visual Studio hides a lot of the complexity of .NET code, and its automatic generation of the code needed to build your application's user interface is a must-have. Most of its features exist to simplify the process of adding code to your application.

Although I will not be including a 20-page review of Visual Studio right here, you will find images of Visual Studio throughout the text, placed so as to advance your understanding of the topics under discussion in each

2. *Datamation*, Volume 29, Number 7, July 1983, pp. 263–265. I also found the text of the article on the Internet by doing a search on the title. A similar version of the text, with only minor editorial changes, also exists under the name "Real Programmers Don't Write Pascal."

8009_ch01.qxd 9/18/06 11:24 AM Page 22

chapter. When you start up Visual Studio for the first time, it displays the "Start Page." (The screenshots in this book are taken from the Professional Edition of Visual Studio 2005.)

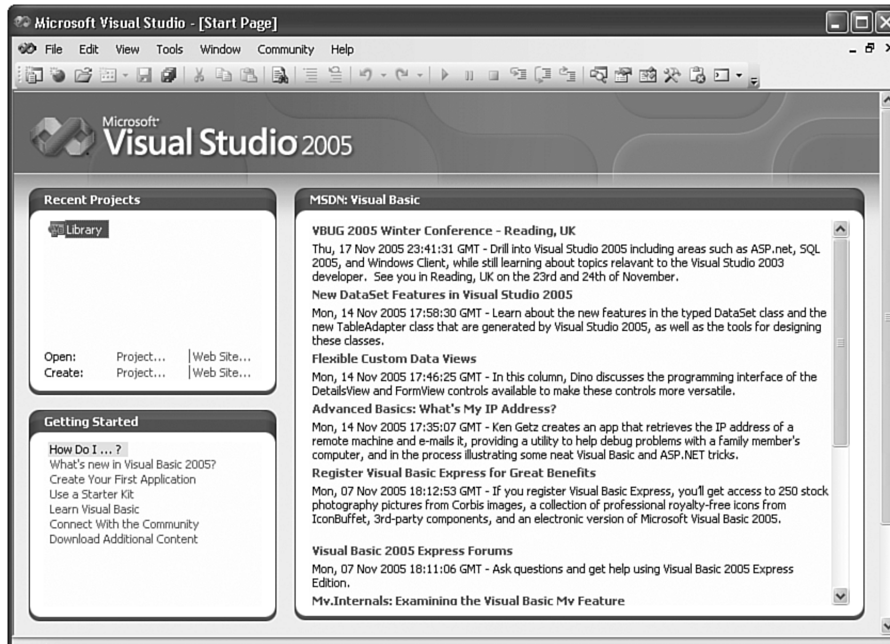

Figure 1-6 The Visual Studio "Start Page"

Visual Studio 2005 is the third major release of the product since .NET's initial introduction in 2002. Each release (in 2002, 2003, and 2005) corresponded to a related release of the .NET Framework (versions 1.0, 1.1, and 2.0, respectively) and of the .NET implementation of Visual Basic. The 2003 release was a relatively minor update to Visual Basic and the Framework, but the 2005 release is major. It is packed with new usability features, and comes in five delicious flavors:

- *Visual Studio 2005 Express Edition.* This entry-level product is geared toward the home hobbyist and weekend programmer who wants to learn .NET and one of its core programming languages, but won't be snuggling up to it on a daily basis. Visual Studio 2005 Express Edition is actually multiple Express Edition language products bundled together, including Visual Basic 2005 Express Edition

8009_ch01.qxd 9/18/06 11:24 AM Page 23

(although Visual Basic 2005 Express Edition is also provided separately). Microsoft's goal is to introduce as many people as possible to the joys of .NET programming, so it offers the Express Edition at no cost. The package includes a simplified Visual-Studio-like user interface, but it does impose a few restrictions on your program-crafting ability. You can still edit the source code directly and craft applications of any complexity, but the Express UI won't always assist you with this. For instance, you cannot develop web applications with the Express product unless you install the separate Visual Web Developer product. Also, Express doesn't include much support for deployment; applications designed with the Express Edition are generally expected to be used on your own workstation only.

- *Visual Studio 2005 Standard Edition.* Visual Studio's Standard Edition is just like the Express Edition, with a few extras thrown in, such as documentation on how to use the BCL and FCL (amazing), and deployment support through the ClickOnce deployment feature. It also includes support for mobile devices, such as cell phones and PDAs.
- *Visual Studio 2005 Professional Edition.* This is the minimum level required by programmers who will develop applications on a daily basis for money. It's the version that I use, and it includes all of the "power" features needed by a single programmer for both desktop and web-based development. The straightjacketed Express user interface is out, replaced by the full Visual Studio "mighty" Integrated Development Environment (IDE). But wait, there's more. You also get SQL Server 2005 Developer Edition. All instructions in this book that relate to using the development environment refer to the Professional Edition. But if you are following along using the Express or Standard Editions, you will be just fine since the interfaces are quite similar.
- *Visual Studio 2005 Tools for the Microsoft Office System.* This "TOS" version is the Profession Edition, but all support for mobile devices is removed, replaced by special components that target the Microsoft Office suite.
- *Visual Studio 2005 Team System.* The *crème de la crème* of the Visual Studio product line is Team System. It includes features needed by development teams that work on projects together, features such as project management tools and source code control. *Visual Studio 2005 Team Foundation Server*, a separate product, can be installed on a shared server, and enhances the features of the Team System package.

8009_ch01.qxd 9/18/06 11:24 AM Page 24

Microsoft is pushing its new version of SQL Server—SQL Server 2005—this time around. An Express Edition is available for entry-level programmers; a Developer's Edition is included in the Visual Studio 2005 Professional Edition and beyond. A special "Everywhere" edition targets mobile platforms. Of course, there's the complete SQL Server product available for full-scale deployments. Microsoft continues to support Microsoft Access, but it is encouraging the use of SQL Server for even small projects due to its tighter integration with .NET (starting with the 2005 release).

Beyond the database support, Visual Studio 2005 has been endowed with several new usability and feature enhancements:

- *Edit and Continue.* This blast from the past was in Visual Basic since version 1.0, but it has been conspicuously absent since the first .NET release in 2002. Edit and Continue allows you to modify Visual Basic source code while actively running and debugging the application within Visual Studio, and continue running the modified application without a restart. The programmers at "M" have surely given their blood, sweat, and tears to this feature, so use it well.
- *Enhanced compile-time warnings and errors.* Visual Studio always flagged invalid statements in your code, but it now flags warnings on code that will compile, and may give unexpected results when executed. Figure 1-7 shows a warning for a declared variable that has yet to be used in code.

```
Public Sub DoSomeWork()
    Dim currentStep As Integer
          Unused local variable: 'currentStep'.
End Sub
```

Figure 1-7 Fair warning

When actual syntax errors appear in code, Visual Studio now makes recommendations on how to fix them (in many cases), and will fix them for you at the click of a mouse button. In Figure 1-8, clicking on the "Insert the missing 'Next'" line in the *Error Correction* window will add in the missing "Next" keyword. If that small red circle and the black arrow to its right look familiar, that's because they're from the "Smart Tags" feature found in Microsoft Office products.

8009_ch01.qxd 9/18/06 11:24 AM Page 25

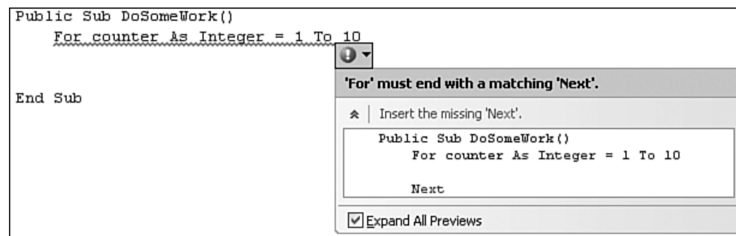

Figure 1-8 Easy error correction

- *ClickOnce Deployments.* This new method of distributing .NET applications imposes fewer requirements on the installing user. For instance, a ClickOnce deployment does not require administrator-level security to install and use the application. Of course, some features may be disabled if the user lacks sufficient privileges.
- *Code Snippets, Project and Item Templates, and Starter Kits.* These features make it easier to integrate pre-written code into your new projects. The Code Snippets feature lets you save a hierarchy of short code blocks for quick insertion into your source code. They include fill-in-the-blank areas if you need them. If you install the source code supplied with this book, you will have a chance to try out Project Templates and Code Snippets, as the samples use those technologies.
- *Generics.* Both the .NET Framework and Visual Basic include support for **generics**, a new feature discussed in Chapter 16, “Generics.” Generics allow you to enforce the use of specific data types on classes that would otherwise impose no such restrictions.
- *Operator Overloading.* Visual Basic adds new support for overloaded operators. This feature lets you assign special meanings to standard language operators, such as the addition operator (+). Instead of adding just numbers together, you develop code to add your own complex classes together; you define what “adding” means for your class.
- *My.* That’s right: just “My.” *My* is a new Visual Basic feature that provides simple and centralized access to FCL features that would normally be spread throughout that class library. You can read more about it in the very next chapter.

8009_ch01.qxd 9/18/06 11:24 AM Page 26

Despite all of these great new features, Microsoft still refuses to implement the most requested Visual Studio feature, "Procedure AutoCompletion," in which Visual Studio would create the entire content of a source code procedure based on your entry of its name and the use of the Control+Space key combination. Instead, they fritter away their time on other so-called productivity features. With Procedure AutoCompletion, you could write entire applications in minutes. Until that feature becomes available, you and I will have to continue writing software, crafting the quality code that users have come to expect from our fingers.

Summary

Over fifteen years ago, Visual Basic transformed the Windows development landscape, with its drag-and-drop programming model, and its glitzy event-driven development structure. But Windows has changed a lot since those days of Windows 3.x. As Windows has changed, Visual Basic has changed right along with it. Visual Basic 2005, through its association with the .NET Framework, provides access to the programming tools needed to development quality applications for the Windows desktop, the Internet, and the next generation of mobile devices.

And Microsoft is not halting this progress with the 2005 release. The next version of Visual Basic, code-named "Orcas," promises to include even more advanced features that will take full advantage of Windows Vista and its .NET Framework 3.0 (formally named *WinFX*) programming interface.

The Project

Welcome to the Project section, the part of each chapter where you have an opportunity to get "hands on" with Visual Studio 2005 and Visual Basic. Development of the Library project, the main project focus of this book, formally begins in Chapter 3, "Introducing the Project," but there's still project work to do in the meantime. In this chapter, I'll introduce you to the sample source code provided with this book, and we'll take a stab at using it.

Since most Project sections, including this one, will involve Visual Studio, make sure you have it installed and ready to use. Also, since each Project section is designed for you to use interactively with the supplied source code, I will assume that you have downloaded and installed the

8009_ch01.qxd 9/18/06 11:24 AM Page 27

source code (see Appendix A, "Installing the Software," for instructions), and are viewing the source code with one eye while you read this section with the other. I will print sections of the source code in the book, but with tens of thousands of source code lines in the Library project, I will not be able to print every line here. You will certainly get a lot out of each Project section by simply reading them, but you will get even more if you have access to the full source code.

In this chapter's project, we'll load a sample program into Visual Studio and run it. There are two ways to do this. The first way is just to open the existing project directly from the installation directory. Browse to the directory where you installed this book's source code, open the "Chapter 1" subdirectory, and double-click the *Chapter1.vbproj* file. This will open the project directly in Visual Studio, ready to use.

The second way is to use the chapter-specific project templates to create new projects in Visual Studio. The Setup program for this book's source code modified your installation of Visual Studio, adding new entries in the *New Project* dialog window. Each of these new "project templates" can be used as the starting point for a new Visual Basic project. To load the Chapter 1 sample program using the template, start Visual Studio. The Start Page will appear, as shown way back in Figure 1-6. From the **File** menu, select **New Project** to display the **New Project** dialog window (see Figure 1-9).

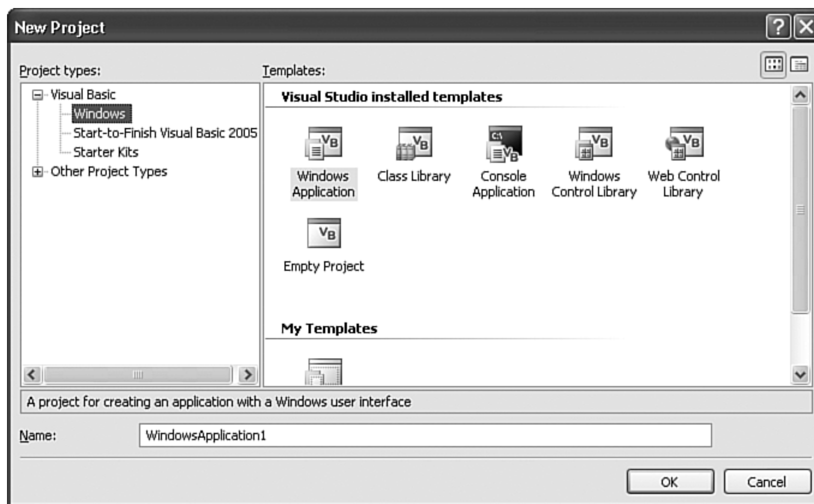

Figure 1-9 The New Project dialog window: So many choices

8009_ch01.qxd 9/18/06 11:24 AM Page 28

Your **New Project** dialog window may differ slightly depending on the features you chose to install with Visual Studio. The available projects are grouped by the description in the **Project types** field. For instance, Figure 1-9 shows the various default project types you can create in Visual Basic, including *Windows Applications* (standard desktop applications for the Windows platform, *Class Libraries* (a DLL of class-defined features), and *Console Applications* (command-line text-based applications). To create a new application, first select the project type, select the **Template** to use, and finally enter the name of the new project in the **Name** field. Clicking the **OK** button creates a new project.

To use the sample Chapter 1 project, select the *Start-to-Finish Visual Basic 2005* entry within the *Visual Basic* project type, and then select *Chapter 1 Sample* from the **Template** field (see Figure 1-10). Finally, click **OK** to create the new sample project.

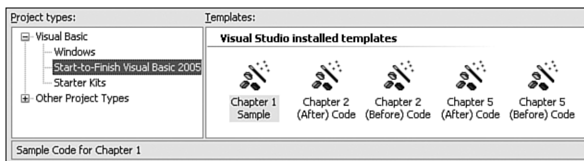

Figure 1-10 Selecting the Chapter 1 Sample project

Once the project loads, access the program's main form by double-clicking on the "Form1.vb" file in the **Solution Explorer** (see Figure 1-11).

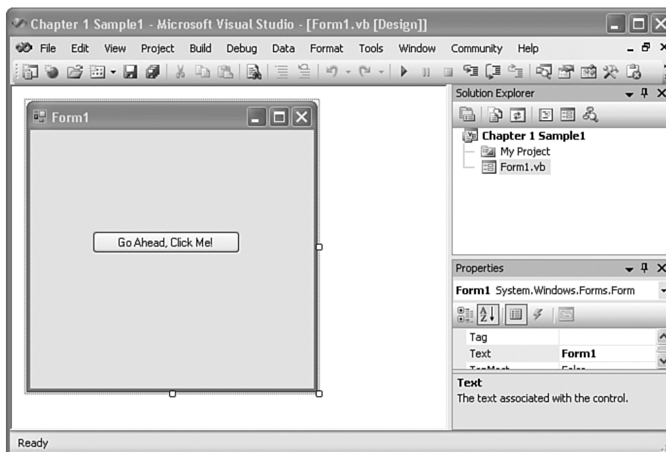

Figure 1-11 The main form of the sample application

8009_ch01.qxd 9/18/06 11:24 AM Page 29

This default presentation of Visual Studio Professional Edition includes three editing components: (1) the main editing area, where the view of “Form1” appears; (2) the **Solution Explorer** panel, which provides access to all files included in the project; and (3) the **Properties** panel, which lets you edit various aspects of the currently-selected item in the main editor area or elsewhere in the user interface.

The sample project is pretty basic. It includes one form with a single action button. Clicking this button in the running application displays a simple message. Run the project by pressing the F5 key. When the main form appears, clicking on the **Go Ahead, Click Me** button displays the message in Figure 1-12 (goal, sweet goal).

Figure 1-12 Hello again, world!

So, how about all of that complex code I had to write to develop this multifaceted application? It’s all there for the viewing. From the **Solution Explorer** panel, right-click on the “Form1.vb” entry, and select **View Code** from the shortcut menu. (As with most source code samples presented in this book, I have had to slightly adjust the code so that it displays properly on the printed page. Generally, this involves splitting a long logical line into two or more shorter ones.)

```
Public Class Form1
   Private Sub Button1_Click(ByVal sender As System.Object, _
         ByVal e As System.EventArgs) Handles Button1.Click
      MsgBox("Hello, World!")
   End Sub
End Class
```

We'll get into the intricacies of such code in later chapters, but here is the gist:

- The main form, **Form1**, is represented in code by a class, named "Form1."
- The form includes a command button named **Button1** that exposes a *Click* event. This event is handled by the *Button1_Click* procedure, a member of the *Form1* class.
- The "event handler," *Button1_Click*, includes a single statement, a "MsgBox" statement. This statement does the heavy lifting by presenting the ever-friendly message box to the world.

That's all of the code that I wrote for "Form1.vb." It sure seems pretty short for all the work it does. There has to be more code hiding somewhere. And sure enough, there are actually half-a-dozen or so more files included in the project. Visual Studio hides these by default, since it manages some or all of the content in these files on your behalf. To view the files, click on the "Show All Files" button (the second toolbar button from the left in the **Solution Explorer** panel). Look at all those files! To see the additional files associated with *Form1*, expand it by clicking on the plus sign to its left (see Figure 1-13).

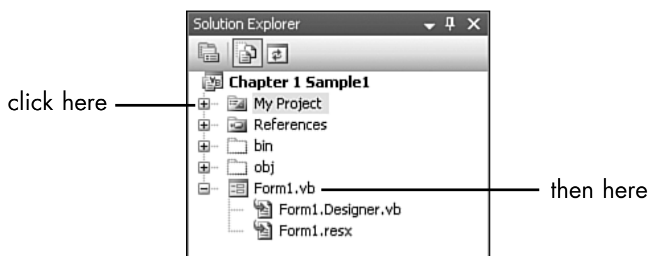

Figure 1-13 Viewing hidden files through the Solution Explorer

Double-click on the "Form1.Designer.vb" entry to see the code that Visual Studio automatically wrote for this form. (Dramatic pause.) Wow! Look at all of that scary code. Actually, it's not that bad. By the end of this book, you will have a firm grasp on all of it. Here in Chapter 1, it's not really necessary to comprehend it all, but there are a few interesting lines to note.

8009_ch01.qxd 9/18/06 11:24 AM Page 31

I'm including line numbers to make it easier to find the matching code in Visual Studio. If you want to view line numbers in Visual Studio (Professional Edition instructions listed here):

1. Select the **Tools ➤ Options** menu item to display Visual Studio's options.
2. Select **Text Editor ➤ Basic ➤ Editor** from the tree-view to the left. If the **Show all settings** field is checked, the last component in the tree-view will be **General**, not **Editor**.
3. Select (check) the **Line Numbers** field on the right.
4. Click **OK** to apply the changes.

If you're new to Visual Basic or .NET programming, don't worry now if all this code doesn't make sense; it will all become clear as you pass through the pages of this book.

```
 1  <Global.Microsoft.VisualBasic.CompilerServices. _
       DesignerGenerated()> _
 2  Partial Public Class Form1

20  <System.Diagnostics.DebuggerNonUserCode()> _
21  Protected Overloads Overrides Sub Dispose _
       (ByVal disposing As Boolean)
```

These lines show *attributes* in action. These two attributes (*DesignerGenerated* and *DebuggerNonUserCode*) are somewhat like the *Obsolete* attribute discussed earlier, in that they provide some informational identity to the related code. *DesignerGenerated* modifies the entire section of *Form1*'s code, while *DebuggerNonUserCode* only modifies the *Dispose* member. For clarity, both attributes include their full namespace paths. The *Global* keyword at the beginning of the *DesignerGenerated* attribute is actually a Visual Basic keyword that says, "Start at the very tippy-top of the namespace hierarchy; this is not a relative path."

```
 2  Partial Public Class Form1
```

Did you see the word *Partial* right there on line 2? I know I did. Hey, wait a minute; "Public Class Form1" also appeared in the *Form1.vb* file, but without the "Partial" keyword. Visual Basic 2005 includes a new feature that lets you divide a single class (*Form1* in this case) among multiple

8009_ch01.qxd 9/18/06 11:24 AM Page 32

source code files by including the "Partial" keyword with at least one of the parts. Pretty cool, eh? It allows Visual Studio to add complex initialization code for your form (as found in this *Form1.Designer.vb* file) without it bothering your main source code file (*Form1.vb*).

```
3   Inherits System.Windows.Forms.Form
```

The *Inherits* keyword defines the inheritance relationship between this new *Form1* class and the previously-written *System.Windows.Forms.Form* class. *Form* is the "base" class, while *Form1* is the "derived" class; *Form1* inherits all of the functionality of the *Form* class, including its initial look and feel. I'll discuss these class relationships in more detail in Chapter 8.

```
44  Friend WithEvents Button1 As System.Windows.Forms.Button
```

Line 44 defines the **Go Ahead, Click Me** button that appears in the center of the form. All controls that appear on your form are separate instances of classes. (*Friend* is a declaration statement described in the next chapter.) The *WithEvents* keyword indicates that this instance of the *Button* class will respond to events, such as a user clicking on it with the mouse. This line doesn't actually create an instance of the *Button* class; that happens back on line 22.

```
22  Me.Button1 = New System.Windows.Forms.Button
```

The *New* keyword creates new instances of classes. In this case, that new instance is assigned to the *Button1* class member defined on line 44. At this moment, *Button1* is a default instance of the *Button* class; it doesn't have any of its custom settings, such as its size and position, or the **Go Ahead, Click Me** display text. All of that is set in lines 27 to 31.

```
27  Me.Button1.Location = New System.Drawing.Point(64, 104)
28  Me.Button1.Name = "Button1"
29  Me.Button1.Size = New System.Drawing.Size(152, 23)
30  Me.Button1.TabIndex = 0
31  Me.Button1.Text = "Go Ahead, Click Me!"
```

8009_ch01.qxd 9/18/06 11:24 AM Page 33

Finally, the button is "glued" onto the form on line 38.

```
38  Me.Controls.Add(Me.Button1)
```

This adds the *Button1* instance to the list of *Controls* managed by *Form1*. The *Me* keyword used throughout this code refers to the *Form1* class itself, so "`Me.Button1`" refers to the *Button1* class member specifically in the current *Form1* class.

Most of the code in this file appears in the *InitializeComponent* member procedure.

```
21  Private Sub InitializeComponent()
       ...
43  End Sub
```

When Visual Basic creates an instance of *Form1* to display on the screen, it calls the *InitializeComponent* procedure to do the work of adding the controls to the form. Actually, Visual Basic calls the form's **constructor**, which in turn calls *InitializeComponent*. Constructors are special class members that perform any needed initialization on a class instance. They are called automatically by .NET each time a class instance is created. In Visual Basic, all constructors use the name **New**, as with the following code:

```
Friend Class ClassWithConstructor
   Public Sub New()
      ' ----- All initialization code goes here.
   End Sub
End Class
```

I'll talk much more about constructors in Chapter 8, but for now, locate the constructor in the code for *Form1*. (Very long pause.) What? There is no constructor? So, if there isn't a constructor, how is the *InitializeComponent* member ever called?

That's what I'd like to know. Actually, when the Visual Basic compiler generates the MSIL code for *Form1*, it adds a constructor silently, a constructor that calls *InitializeComponent*. How about that! Why didn't Microsoft simply include the constructor's code right in the source code?

8009_ch01.qxd 9/18/06 11:24 AM Page 34

It's a simplicity-for-the-programmer thing. They needed to have a default constructor that would call *InitializeComponent*, but they didn't want a conflict to arise if you added your own default constructor in the non-Designer file. So they hid all of the code until it came time to actually compile the form. Clearly, it's all rather hush-hush, so let's move on.

Well, that's pretty much the entire code, at least the part that matters to us now. Although we will rarely, if ever, examine the Visual Studio-generated code for the forms in the Library project, it's good to see what's going on behind the scenes. If you were a Visual Basic 6 programmer, you probably looked at the source code for your forms through Notepad at one time or another. If you did, you noticed that the form and all of its controls were defined with a hierarchy of special commands, and not with actual Visual Basic code. In .NET, that's all changed; the form and all of its controls are created with ordinary Visual Basic code, so you can access it all and see what is really going on.

Now, turn to Chapter 2, "Introducing Visual Basic," where I delve into the Visual Basic language itself.