CHAPTER 1

Understanding Visual Basic .NET

- The .NET Framework
- The .NET languages
- Command-line compilation
- Visual Studio .NET requirements

This chapter explains important background material related to Visual Basic .NET (VB .NET). If you are the hasty type and want to dive right into programming, there's no need to start with this chapter. (Although you may want to check the "Visual Studio .NET Requirements" section at the end of the chapter, to make sure you have all the tools you need.) You can skip right over it and go on to Chapter 2, "Introducing Projects, Forms, and Buttons." (The sample project in the "For the Very First Time" section of Chapter 2 will get you up and running with a Windows program in the blink of an eye.) As you go on with VB .NET, understanding some of the concepts behind it will become important. You can then come back to this chapter for a dose of background information.

On the other hand, if you prefer to have your ducks in a row before you get to programming with VB .NET, you can start right here. Armed with the conceptual understanding provided in this chapter, you should be able to make good progress as a VB .NET programmer.

The chapter includes an important preliminary topic: command-line compilation. You would probably be pretty foolish to try to create Windows or web programs outside the world-class Visual Studio integrated development environment (IDE). But it's important to know that you can, and that Visual Studio, which is used throughout the rest of this book, is optional. You should clearly understand that VB .NET (the language) is complementary but not identical to Visual Studio .NET (the IDE).

What Is .NET?

The term *.NET* is somewhat confusing because it is commonly used in a variety of contexts. Some of these contexts are mostly marketing in nature, rather than the kind of precise language needed when attempting to understand technology. Here are some of the ways in which .NET has been used:

- To mean the .NET Framework, a runtime platform and programming framework largely consisting of class libraries
- To include Visual Studio .NET, a professional IDE optimized for working with .NET Framework languages, including VB .NET and C# .NET
- To refer to .NET Enterprise Servers, a set of enterprise server products such as Biztalk Server, Exchange Server, Mobile Information Server, and SQL Server, which have been given the .NET moniker for marketing purposes
- To describe .NET My Services, also sometimes called Hailstorm, which is a vision for creating services—such as lists, contacts, schedule information, and more—that can be accessed in a platform- and language-independent way

This book is mostly concerned with the first two of these meanings: the .NET Framework and Visual Studio .NET. The focus in this chapter is understanding the .NET Framework.

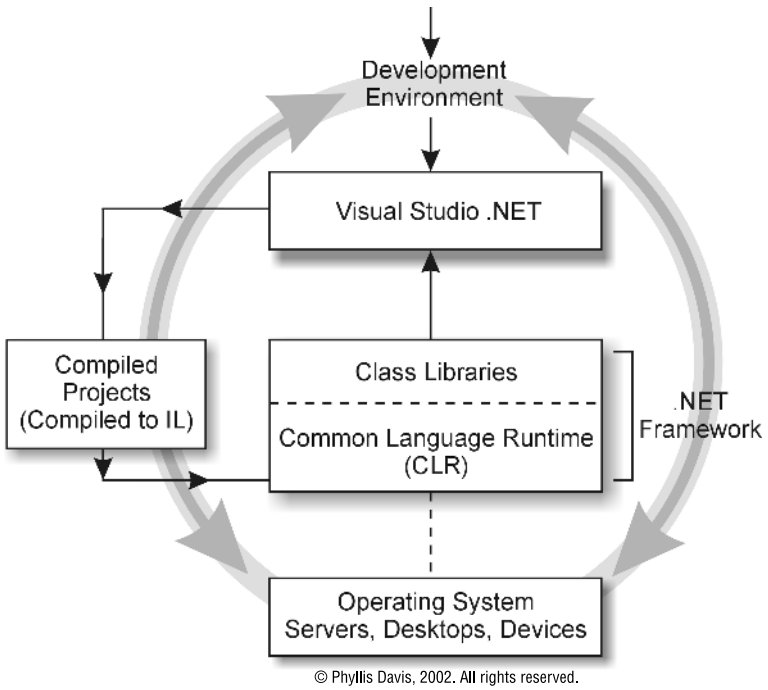
The .NET Framework

The .NET Framework consists of two main parts:

- The Common Language Runtime (CLR), which is a platform that manages code execution (discussed in detail in the next section)
- The .NET Framework class libraries

The relationship of the two parts of the .NET Framework to the operating platform and to Visual Studio .NET is shown in Figure 1.1.

FIGURE 1.1:
The two parts of the .NET Framework have different functions: the Framework class libraries are used to build applications, and the CLR layers on top of the operating system manage execution of a program.



As you'll see in the "Using a Text Editor to Create VB .NET Programs" section later in this chapter, it's possible to create, compile, and run VB .NET programs without using Visual Studio .NET. However, as programs get more complex, this quickly becomes cumbersome. Except for the one example in this chapter, this book will show you how to create VB .NET applications using Visual Studio, because it's the simplest way to go most of the time. But it's important that you be aware that Visual Studio is not integral to the definition of the .NET Framework.

The .NET Framework Base Class Library is a large set of types, classes, and interfaces that form the basis, or blueprint, for objects that can be used programmatically to create applications, including Windows forms, web forms, and web services applications. The .NET Framework types form the basis for building .NET applications, components, and controls. .NET Framework types perform many functions, such as representing value types for variables, performing input-output (I/O) operations, and providing data access. An example of a value type that is probably familiar to most programmers is the Integer, which in the .NET Framework is called an Int32 and used to type values that contain a signed 32-bit integer number.

NOTE The .NET languages are interoperable, which means that one language can use class libraries in another. For example, a VB .NET program can use a class developed in C# .NET, or vice versa. To ensure this interoperability, the .NET Framework types are compliant with the Common Language Specification (CLS). They can be used by any language that conforms to the CLS.

The .NET Framework uses a dot operator (.) syntax to designate hierarchies. Related types are grouped into namespaces, so that they can be more easily found. (See the "Namespaces" section later in this chapter for more details.)

Reading left to right, the first part of a type, up to the first dot, is the namespace name. The last part of the name, to the right of the final period, is the type name. For example `System.Boolean` designates a Boolean value-type in the `System` namespace. `System.Windows.Forms.MessageBox` designates the MessageBox class with the `Forms` namespace, which is part of the `Windows` namespace, which is part of `System`.

TIP When you are referring to a member of the `System` namespace, you can usually leave off `System`. So, for example, the variable declaration `Dim IsEmployee As Boolean` is the equivalent of `Dim IsEmployee As System.Boolean`.

As these examples suggest, the `System` namespace is the root namespace for all types within the .NET Framework. All base data types used by all applications are included in the `System` namespace or the `Microsoft` namespace.

One of the most important types within the `System` namespace is `System.Object`. `System.Object`, also called the Object class. The Object class is the root of the .NET type hierarchy and the ultimate parent (or *superclass*) of all classes in the .NET Framework. This implies that the members of the Object class—such as `GetType()` and `ToString()`— are contained in all .NET classes.

NOTE Don't let this talk of objects, types, classes, and hierarchies throw you! It all works out in a fairly intuitive way once you start programming. But if you want to jump ahead, the Object Browser, the best tool for learning about class hierarchies, is explained in Chapter 14, "Using the Object Browser." Object-oriented programming concepts, including some of the terminology used in this section, are explained and further defined in Chapter 15, "Object-Oriented Programming in VB .NET."

The Common Language Runtime (CLR)

The CLR manages execution of compiled .NET programs. The role of the CLR is comparable to Sun's Java Virtual Machine (JVM) and the VB runtime library that shipped with older versions of VB.

The CLR is a runtime for all .NET languages. Its job is to execute and manage all code written in any language that has targeted the .NET platform.

When you write an application in .NET, you can choose from a number of languages. Primarily these are VB and C# (pronounced "see sharp"). You can also build .NET applications using languages such as COBOL, Eiffel, Pascal, and even Java. Each of these languages will include its own compiler, written by third parties. However, instead of compiling into machine code, as compilers typically do, the language-specific just-in-time (JIT) compiler translates the code (whether it is VB, C#, or any other language) into another language called Microsoft Intermediate Language (MSIL, or even just IL for short). The IL is what you actually deploy and execute.

Upon execution, the CLR uses another compiler to turn the IL into machine code that is specific to the platform where it's running. In other words, the CLR may compile the IL into one thing on a machine that runs on an AMD processor and into something else on a machine with a Pentium processor.

TIP Intermediate Language (IL) is, in fact, fairly easy to read and understand. You'll find that documentation for doing this is part of the .NET Framework Software Development Kit (SDK). In addition, you may find it interesting to know about a program named `ILDasm.exe`, which ships with the .NET framework. ILDasm is an Intermediate Language disassembler. You can run any .NET compiled program through ILDasm, including important parts of the .NET Framework itself. ILDasm will output the IL code that has been created, along with other information, including namespaces, types, and interfaces used.

The CLR handles, or manages, some other very important aspects of the life of a program:

Type, version, and dependency information .NET compilers produce not only IL, but also metadata that describes the types contained within an EXE or DLL and version and dependency information. This means that the CLR can resolve references between application files at runtime. In addition, the Windows system Registry is no longer needed for keeping track of programs and components. One way of thinking of the CLR is as an object-oriented replacement for the Win32 Application Programming Interface (API) and Component Object Model (COM, used for component interoperability). Placing the CLR as a kind of abstraction layer on top of Windows has neatly solved a great many of the technical problems with Windows programming. This approach has worked to the benefit of VB programmers, since much of the Win32 functionality was not available to us. All of the functionality of the CLR and its class libraries are exposed as objects and classes, whose methods can be used in your programs.

Garbage collection *Garbage collection* means that memory is automatically managed. You instantiate and use objects, but you do not explicitly destroy them. The CLR takes care of releasing the memory used by objects when they are no longer referenced or used. It is understandable that some programmers like to manage memory themselves, but this practice, particularly in large team projects, inevitably produces memory leaks. The CLR's automatic garbage collection solves the problem of memory leaks.

NOTE There is no way to determine when the CLR will release unused memory. In other words, you do not know when an object will be destroyed (also called *nondeterministic* finalization). To say the same thing in yet another way, just because there are no more in-scope references to an object, you cannot assume that it has been destroyed. Therefore, you cannot place code in an object's destructor and expect deterministic execution of the code.

Code verification Code verification is a process that takes place before a program is executed. It is designed to ensure that a program is safe to run and does not perform an illegal operation, such as dividing by zero or accessing an invalid memory location. If the program does include code that does something naughty, the CLR intercepts the flawed commands and throws an exception before any harm can be done.

Code-access security Code-access security lets you set very granular permissions for an application based on "evidence." For example, you can configure an application that is installed on the local machine to access local resources such as the filesystem, Registry, and so on. However, the same application, if run from the intranet, can be denied those permissions. If it tries to perform an operation for which it does not have permissions, the CLR will prevent the operation.

Managed Code

The first step in the managed-code process is to compile a .NET project to IL, which also generates the required metadata. At execution, a JIT compiler translates the IL code to native machine code. During this translation process, the code is also verified as safe. This verification process checks for the following:

- Any reference to a type is strictly compatible with the type.
- Only appropriately defined operations are performed on an object.
- Identities are what they claim to be.
- IL code is well-formed.
- Code can access memory locations and call methods only through properly defined types.

If a program fails this verification process, an exception is thrown and execution is halted. Otherwise, the CLR provides the infrastructure that actually executes the native code.

The managed-execution processes, and services provided by the CLR such as memory garbage collection, are collectively referred to as *managed code*.

Programming in the .NET Framework

Fundamentally, programming in the .NET Framework means making use of the classes, objects, and members exposed by the Framework, building your own classes on top of these, and manipulating the resulting objects using familiar programming language and syntax. (If you are unfamiliar with programming languages altogether, don't worry—starting in Chapter 2, each step will be explained as we go along.)

NOTE One of the goals of the .NET Framework is to make programming more standardized across languages. Thus, you can create and use objects based on the same classes, whether you are programming in VB .NET, C# .NET, or Managed C++.

This section explains some of the key building blocks and concepts of the .NET Framework that will be helpful for you to understand as you begin creating VB .NET programs: assemblies, namespaces, and objects.

Assemblies

Assemblies are the fundamental unit for deployment, version control, security, and more for a .NET application. Every time you build an executable (EXE) or a library (DLL) file in .NET, you are creating an assembly. An assembly contains the information about an application that used to be stored in a type library file in previous versions of VB, and you use the contents of assemblies and add references to them much as you would manage a type library.

The Assembly Manifest

When you start a new VB project, it is the basis of an assembly. Within each built assembly is a manifest, which is part of the executable or library. In VB .NET, some of the general manifest information is contained in a file that is part of the project named `AssemblyInfo.vb`. Figure 1.2 shows a small project in the Visual Studio Solution Explorer with `AssemblyInfo.vb` selected, and Figure 1.3 shows the contents of a sample `AssemblyInfo.vb` file when opened with the Visual Studio editor.

TIP To open the `AssemblyInfo.vb` module, double-click it within the Solution Explorer. Using the Solution Explorer is covered in Chapter 2.

FIGURE 1.2: Each VB .NET project includes a file that is the assembly manifest.

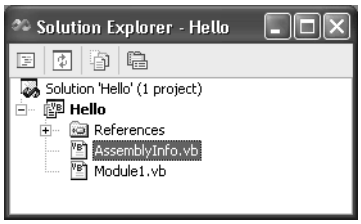
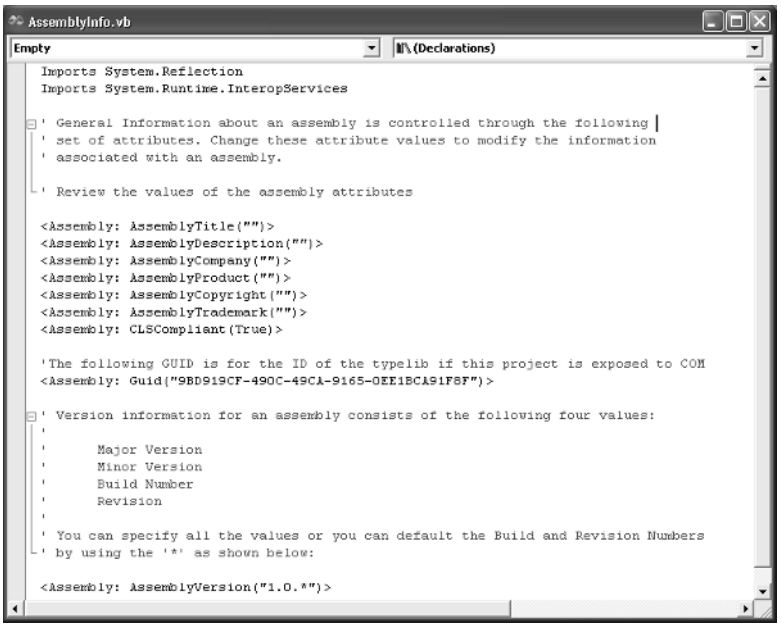


FIGURE 1.3: The assembly manifest contains information about content, version, and dependencies, so that VB .NET applications do not depend on Registry values to function properly.



The assembly manifest can be thought of as a table of contents for an application. It includes the following information:

- The assembly's name and version number
- A file table listing and describing the files that make up the assembly
- An assembly reference list, which is a catalog of external dependencies

The external dependencies in the assembly reference list may be library files created by someone else, and likely some of them are part of the .NET Framework.

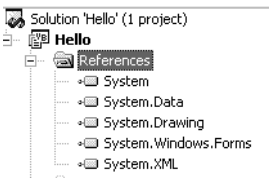
Assembly References

To use an assembly, or an object within an assembly, it must be referenced in your project. Depending on the type of project, you'll find that many of the assemblies that are part of the .NET Framework are referenced by default.

Different project types have different default references. The references that come "out-of-the-box" for a Windows forms project are not the same as those for a web forms project, although both do reference certain important .NET assemblies such as `System.dll`.

You can see which assemblies are already referenced in a project by expanding the References node in the Solution Explorer, as shown in Figure 1.4.

FIGURE 1.4: You can view the references in a project in the Solution Explorer.



If you need to reference an assembly that is not already included in your project, follow these steps:

1. Open the Visual Studio Project menu and click Add Reference (select Project ➢ Add Reference). The Add Reference dialog will open, as shown in Figure 1.5.
2. Click the Browse button in the upper-right corner of the Add Reference dialog. The Select Component dialog will open.
3. Locate the assembly to be added and click Open. The assembly will be added to the Selected Components panel of the Add Reference dialog.
4. Click OK to add the reference to your project.

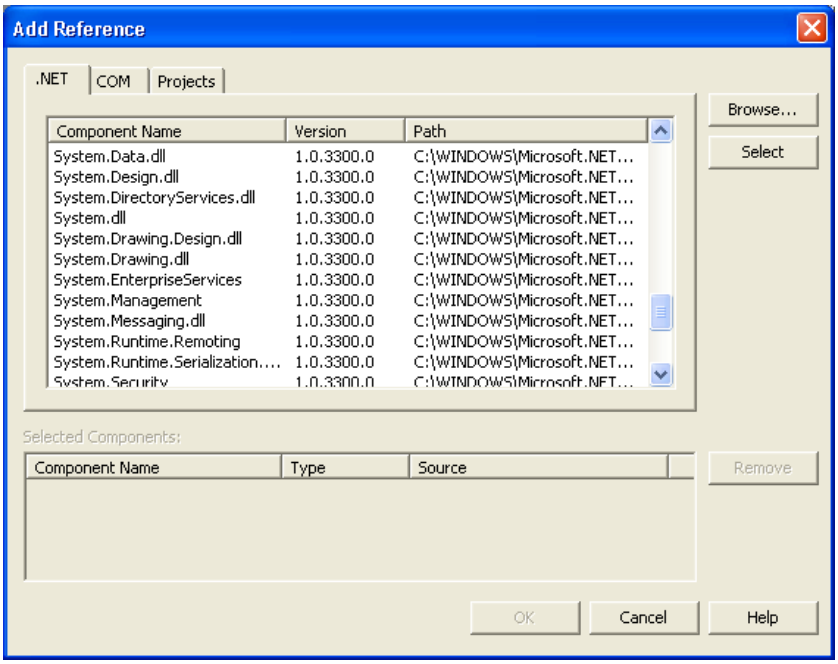


FIGURE 1.5:
The Add Reference dialog is used to add a reference to a project.

Once a reference to an assembly has been added to a project, you'll need to reference an item you want to use using the namespace it is in. You'll find more information about working with references in Chapter 15, "Object-Oriented Programming in VB .NET."

Namespaces

Namespaces are used to organize the objects (such as classes) within an assembly. Assemblies can contain many namespaces, which, in turn, can contain other namespaces. Namespaces are used to make it easier to refer to items, to avoid ambiguity, and to simplify references when large groups of objects are involved (for example, in a class library).

By default, every executable file you create in VB .NET contains a namespace with the same name as your project, although you can change this default name.

You should also know that namespaces can span multiple assemblies. In other words, if two assemblies both define classes within a namespace `myspace`, then the `myspace` namespace is treated as a single set of names.

Namespace References

There are several ways to refer to an item within a namespace once the assembly containing the item you are interested in has been referenced. You can use the *fully qualified* name of the item, as in this example:

```
Dim myBox As New System.Windows.Forms.TextBox
```

Alternatively, you can place an `Imports` statement at the beginning of a code module, as shown here:

```
Imports System.Windows.Forms
```

After you add an `Imports` statement, all of the names in the imported namespace can be used (provided they are unique to your project), like this:

```
Dim myBox As New TextBox
```

Important VB .NET Namespaces

Table 1.1 lists some of the namespaces that are important to VB .NET developers. For more information about creating and using namespaces, see Chapter 15.

TABLE 1.1: Selected .NET Framework Namespaces

Namespace	Description
`Microsoft.VisualBasic`	Contains the runtime used with the VB .NET language, as well as classes that support VB .NET compilation and code generation
`System`	Contains fundamental classes that define types, events, event handlers, interfaces, data-type conversion, mathematics, and much more
`System.Collections`	Includes a set of classes that lets you manage collections of objects
`System.Data`	Includes the classes that comprise the ADO.NET architecture
`System.Diagnostics`	Provides classes used for debugging, tracing, and interacting with system processes, event logs, and performance counters
`System.Drawing`	Provides access to GDI+ basic graphics functionality (namespaces hierarchically beneath `System.Drawing`—including `System.Drawing.Drawing2D` and `System.Drawing.Text`—provide more advanced and specific GDI+ graphics functionality)
`System.IO`	Contains types used for reading and writing to data streams and files
`System.Reflection`	Contains classes and interfaces that provide type inspection and the ability to dynamically bind objects
`System.Web`	Contains the classes that are used to facilitate browser-server communication and other web-related functionality
`System.Web.Services`	Contains the classes used to build and consume web services
`System.Windows.Forms`	Contains the classes for creating a Windows-based user interface
`System.XML`	Provides support for processing XML

Objects and Classes

It's important to understand the distinction between objects and classes. An *object* is a unit of code and data created using a *class* as its blueprint. Each object in VB .NET is defined by a class, which specifies the properties, methods, and events—collectively referred to as *members*—of the objects based on the class.

Objects, which can themselves contain other objects, are manipulated by the following:

- Setting and retrieving property values
- Invoking object methods
- Executing code when an object event has occurred

Once you have defined the class, you can create as many objects as you need based on the class. The process of creating an object based on a class is called *instantiation*.

A metaphor that is often used is that of cookie cutters and cookies. The cookie cutter is the class, and it defines the characteristics of the cookie, such as its size and shape. Each cookie is an object based on the cookie-cutter class.

Each object, which is called an *instance* of a class, is identical to other objects based on the same class when it is created. Once objects exist, they will likely be loaded with different values than other instances of the same class. The class might specify that each Employee object has Name and Salary properties. Once the Employee objects are instantiated, each will probably have a distinct name and may have a different salary.

The controls in the Toolbox in VB .NET are representations of classes. When a control is dragged from the Toolbox to a form, an object that is an instance of the control class is created. (See Chapter 2 and Chapter 7, "Working with Windows Form Controls," for information about working with the Toolbox.)

A form, which represents an application window that you work with at design time, is a class. When you run the project containing the form, VB .NET creates an instance of the form's class. (The following chapters explain the use of Windows forms in detail.)

When one class inherits from another class—which can be done in code using the `Inherits` keyword or, in some circumstances, by using visual inheritance—the blueprint for the members of the parent class is now transferred and becomes the blueprint for the members of the newly created child class. With this inherited blueprint as a starting place, the members of the class can be extended and changed, and new members can be added. As I mentioned earlier in this chapter, the `System.Object` class is the ancestor, or *superparent*, of almost all classes you will use in VB .NET.

Programming in VB .NET is an exercise in the use of objects and classes, so you will find information about them and how to work with them in every chapter of this book. For information about the theory and background of object-oriented programming, see Chapter 15.

The .NET Languages

Visual Studio .NET is specifically designed to help programmers develop applications in VB .NET, C# .NET, and C++. VB .NET and C++ are the next generation of languages that have been in use for years. C# .NET is an entirely new language developed for .NET. Since much of the .NET Framework is written in C#, it is in some sense the "native" language of .NET.

NOTE Programs written in VB .NET and C# .NET produce CLR-managed code. (An exception to this is if an old-style COM component is added to a project.) Programs written in C++ produce unmanaged code, unless the C++ Managed Extensions are used (by selecting a Managed C++ project type).

In addition to these three core development languages, it is expected that programmers will use Visual Studio for working with ancillary language tools, including the following:

HTML The ASP web forms editor provides excellent Hypertext Markup Language (HTML) support. See Chapter 19, "ASP.NET Web Applications" for details.

JScript JScript is essentially JavaScript and is used in web applications (see Chapter 19 for an example) and for some kinds of application customization.

SQL Structured Query Language (SQL) is used for interacting with databases. Visual Studio provides tools for automatically generating SQL, as explained in Chapter 17, "Working with Data and ADO.NET."

XML eXtensible Markup Language (XML) is used to describe structured data, and .NET uses XML as a primary technology for interoperability. See Chapter 18, "Working with XML in VB .NET," for details.

Upcoming .NET Languages

A number of third parties (companies other than Microsoft) are creating CLS-compliant versions of languages that target the CLR and the .NET platform. Time will tell whether any of these other languages—which range from research languages to products intended for commercial development—gain any traction. However, the very existence of these languages speaks to the scope of the ambition of the .NET platform. .NET languages in the works include the following:

- APL
- COBOL
- Eiffel#
- FORTRAN
- Mondrian

- Oberon
- Perl
- Python
- Smalltalk

From VB6 to VB .NET

Some VB users will feel that VB .NET is so different from Visual Basic version 6 (VB6) that it counts as an entirely new language. There is some truth to this, because many things have changed. But another way of looking at the transition to VB .NET is that you can go on writing VB code in very much the way you always have.

TIP In order to make it easier for VB6 programmers, Microsoft has provided a whole special set of classes in the `Microsoft.VisualBasic.Compatability.VB6` namespace.

Readers who are new to VB need not worry about how VB .NET differs from its predecessors. They will find an intuitive, easy-to-use, robust language that is fully object-oriented (as explained in Chapter 15). The language can be strongly typed, which is the best programming practice, or perform implicit type conversions, if you prefer. VB .NET is a true peer to the other .NET languages; there's really no reason to choose VB over C#, or vice versa, other than personal preference.

The CLR/.NET Framework is a comparable mechanism to the JVM, although undoubtedly, the extent to which .NET is deployed on platforms other than Windows will be limited by technology and industry politics.

If you are a VB6 programmer interested in migrating applications from VB6 to VB .NET, you'll find some useful information in Appendix B, "Migrating Applications from VB6 to VB .NET." You'll also find information about some of the language differences between VB6 and VB .NET in Appendix C, "Key Syntax Differences Between VB6 and VB .NET."

The History of Visual Basic

The BASIC (Beginner's All-Purpose Symbolic Instruction Code) programming language was invented in the early 1960s by two Dartmouth College professors, John G. Kemeny and Thomas Kurtz. They wanted to create a language that was good for teaching computer programming, and they succeeded wildly in meeting that goal.

From its earliest years, BASIC was very easy to understand because it is English-like and unstructured—meaning not too fussy about how you organize programs and how you type variables. (Of course, this is no longer true about VB .NET!)

Continued on next page

A trade-off for this ease of use was speed of execution. Early versions of BASIC were slow because they were interpreted (translated into machine code on the fly), rather than compiled (run as a stand-alone program that has already been converted to machine code). The current incarnation, VB .NET, is really "neither fish nor fowl" in this respect, since it is just-in-time compiled, but its performance characteristics are really quite good.

Microsoft has a long history of commitment to BASIC (and the languages that descended from BASIC). In the 1980s, Microsoft shipped various versions of the BASIC language, such as QuickBasic (shipped in 1982) and QBasic, part of the MS-DOS 6 product.

In the early 1990s, as Microsoft Windows appeared on the scene, a visual version of BASIC, Visual Basic 1.0, was created, leaning heavily on the concepts originated by interface designer Alan Cooper, who has been called the father of Visual Basic (VB). VB added an intuitive visual framework for creating an application's interface and a straightforward mechanism for responding to events to the easy-to-use, unfussy underlying BASIC language.

Possibly on the principle that "anything this easy can't really be good," VB got a reputation as the Rodney Dangerfield of languages—a toy development environment, not really suitable for serious work, and not worthy of respect. VB programmers responded by pointing out how much more productive they were using their "toy" language than their hardcore cousins.

By the late 1990s, millions of programmers were using VB, now VB6, more than any other language. However, its future had become murky. Java, a new language written from the ground up under the sponsorship of Sun Microsystems, was gaining currency as truly object-oriented and cross-platform (provided the platform had a Java Virtual Machine). And the relationship between C++ programmers and VB programmers had settled into a situation in which C++ programmers wrote the heavy-lifting components and VB coders wrote the user interfaces that connected to the C++ components. As the time lengthened since the last release of VB (VB6 came out at the end of 1997), the future of VB became uncertain. VB .NET is the answer to these concerns.

Using a Text Editor to Create VB .NET Programs

As I've mentioned earlier in this chapter, it's important to understand that the VB .NET language is distinct from the Visual Studio .NET development environment. Visual Studio happens to be by far the best and easiest way to create VB .NET programs. I can't really imagine anyone trying to develop a complex Windows application using Notepad, but, in theory, it could be done, and that is the point.

In this section, you'll see how to create a simple VB .NET application using Notepad. Before we get started, you should know about *console applications*. Console applications typically have no user interface other than printed text on the screen. They are run from a

command line, with input and output information being exchanged between the command prompt and the running application.

TIP You can create a console application from within VB .NET by selecting Console Application as the project type from the New Project dialog.

As an example, we'll use Notepad to create a VB .NET program that prints "Hello, World!" to the screen and pauses. Next, we'll compile using the VB .NET command-line compiler, `vbc.exe`. Finally, we'll run the program in the console window.

Creating a VB .NET Program in Notepad

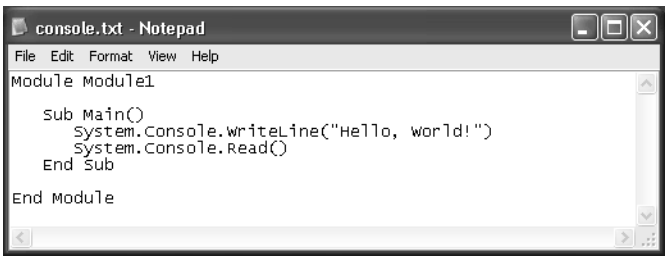
To create our sample program, open Notepad. Type in the program shown in Listing 1.1.

Listing 1.1 A "Hello, World!" Console Application

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

When you're finished, save the file as `console.txt`, as shown in Figure 1.6.

FIGURE 1.6: The code to be compiled is saved as a Notepad text file.



Let's take a look at some of the individual lines in Listing 1.1. The first line names the module:

```
Module Module1
```

The module is named `Module1`, but could instead be named anything you like.

The next line designates the subroutine:

```
Sub Main()
```

This is the procedure that is the program's entry point, or where it starts executing.

Following `Sub Main()` is the code to write the text:

```
System.Console.WriteLine("Hello, World!")
```

This line uses the `WriteLine()` method of the `System.Console` object to display a line of text on the screen.

The last line in the subroutine keeps the text on the screen:

```
System.Console.Read()
```

This uses the `Read` method of the object to pause things so that the text stays on the screen until you hit a key.

As you can see, you can't have a much simpler program in VB .NET—or in any other language!

Compiling the Program

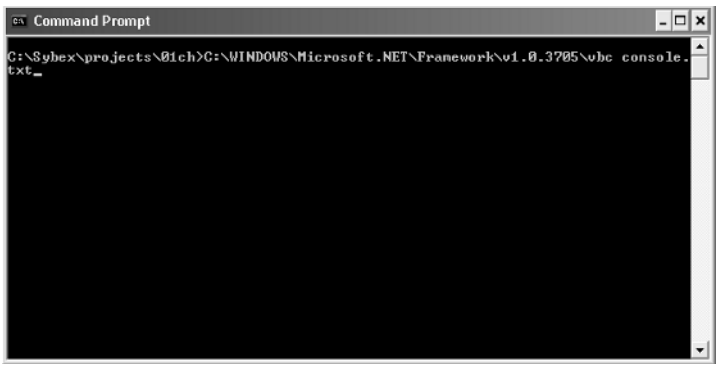
To compile the program, open a command window. Depending on your system, you probably have a version of `cmd.exe` available in the `Windows\System32` folder. You'll probably also find a link that opens a command window on the Windows Program ➢ Accessories menu. In addition, Visual Studio provides a command prompt window, which you can open from the Windows programs menu by selecting Microsoft Visual Studio .NET ➢ Visual Studio .NET Tools ➢ Visual Studio .NET Command Prompt.

TIP If you use the Visual Studio .NET Command Prompt, `vbc.exe` will already be in your path, which means that you don't need to know its location or type in its full path to invoke it.

With a command window open, invoke the VB .NET compiler, `vbc.exe`, with **`console.txt`** as the argument, as shown in Figure 1.7. (You'll find `vbc.exe` in one of the folders beneath `Windows\Microsoft. net\framework`.) For example, the command line to compile `console .txt` might look like this:

```
c:\windows\Microsoft.net\framework\v1.0.3705\vbc console.txt
```

FIGURE 1.7: You can use vbc.exe, the command-line VB .NET compiler, to create executable programs.



Press Enter. A compiled executable named `console.exe`—the original filename without the suffix and with `.exe` added as the filename extension—will be created.

VB Compiler Switches

There are a number of command-line switches you can use with `vbc.exe`. The `/out:`*`filename`* option names the executable (as opposed to the default described in this example).

The `/target` option allows you to specify the type of the output file:

- `/target:exe` produces a console application executable.
- `/target:winexe` produces a Windows executable.
- `/target:library` creates a DLL.

For a full list of VB .NET command-line compiler switches, see the "Visual Basic Compiler Options" topic in online help.

Running the Application

Run the new application, `console.exe`, either from the command line or by double-clicking it in Windows Explorer. The text "Hello, World!" will be displayed in the console.

You've now successfully created, compiled, and run a VB .NET program without using Visual Studio. It's true that this program doesn't do much. It's also true that in the real world, you'll probably almost always use Visual Studio for creating and compiling your programs. But now you know, for once and for all, that the programming language is not the development environment—an important insight.

Visual Studio .NET Requirements

Since we'll be using Visual Studio .NET in the remainder of this book, you'll need to make sure that your system meets the requirements for running it. Let's review the software and hardware necessary for the projects covered in this book.

Software Requirements

Obviously, you'll need a copy of Microsoft's Visual Studio .NET, including the .NET Framework. You can purchase Visual Studio .NET online (and download the product) at `http://msdn.microsoft.com/vstudio/`.

Also, your computer should be running (in either their Professional, Server, or Advanced Server guises) Microsoft Windows NT 4.0 (with service pack 6A) or Windows 2000 or XP, with the latest service packs. Note that NT 4.0 Workstation is supported only for client-side development.

Finally, you will need a copy of Microsoft's web server, Internet Information Server (IIS) version 5.0 or later, and Internet Explorer 6.0 or later.

TIP If you don't have Internet Explorer 6.0, it will be installed when you install Visual Studio .NET.

Hardware Requirements

You will need a computer powerful enough to run your particular mix of software and operating system. Microsoft's minimum recommended specifications for Visual Studio .NET are a 450MHz Pentium II class processor or above, 64MB of RAM, and a 3GB of installation space available on the hard drive. Microsoft's recommended specifications are a 733MHz Pentium III class processor, 128MB of RAM, and a 3GB hard drive for the installation.

As a practical matter, Visual Studio .NET will run very poorly on a system that is as low in resources as even the Microsoft-recommended system. If at all possible, I suggest using a system with a 733MHz Pentium III processor, at least 256MB of RAM, and at least a 10GB hard drive.

Summary

Enough preliminaries! VB .NET is an exciting and elegant language. Familiar because of its antecedents, in this incarnation, VB really flies! The .NET Framework is a powerful and radical solution to many development problems.

This chapter has provided background information that will help you to learn the language faster and be a better VB .NET programmer. Now, let's get started building programs!