



## CHAPTER 1

---

# *Introduction*

1.1	Developing for the .NET platform	2	1.4	Exploring the .NET Framework class library	7
1.2	A first .NET program	4	1.5	Putting .NET to work	11
1.3	The platform vs. the programming language	6	1.6	Summary	14

Since Microsoft unveiled .NET in the summer of 2000, many have had difficulty defining exactly what .NET is. According to Microsoft, “.NET is Microsoft’s platform for XML Web Services.” That’s true, but it is not the whole story. Here are a few of the highlights:

- .NET is a new platform for the development and deployment of modern, object-oriented, “managed” applications.
- Fully functional .NET applications can be developed using any programming language that targets the .NET runtime.
- .NET provides a comprehensive framework of language-neutral class libraries.
- .NET supports the creation of self-describing software components.
- .NET supports multilanguage integration, cross-language component reuse, and cross-language inheritance.
- .NET introduces a new way to develop Windows desktop applications using the Windows Forms classes.

- .NET provides a new way to develop Web browser-based applications using the ASP.NET classes.
- .NET's ADO.NET classes provide a new disconnected architecture for data access via the Internet.
- .NET supports the creation of platform-independent XML Web services using standards such as SOAP (Simple Object Access Protocol) and WSDL (Web Service Description Language).
- .NET provides a new architecture for the development and deployment of remote objects.
- .NET makes many Windows technologies and techniques obsolete.

So .NET is big, and requires almost a fresh start for developers working with Microsoft platforms and tools. For Microsoft, the release of .NET is arguably the most important event since the introduction of Windows itself.

## 1.1 DEVELOPING FOR THE .NET PLATFORM

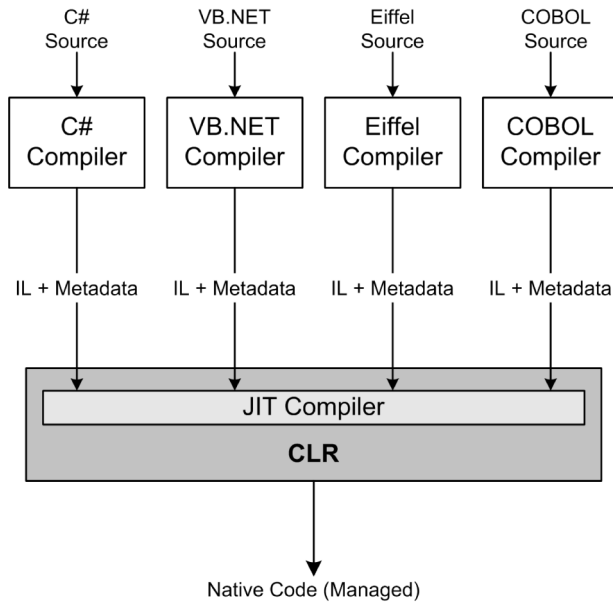
For Windows developers, .NET offers relief from the hegemony of Visual C++ and Visual Basic. .NET is independent of any programming language. There are .NET compilers available for several languages and more are planned. Available at the time of writing are C#, Visual Basic .NET, JScript .NET, COBOL, Perl, Python, Eiffel, APL, and others. You can also use the *managed extensions* for Visual C++ to write .NET applications. .NET supports these languages by supporting none directly. Instead, .NET understands only one language, Microsoft Intermediate Language (IL).

### 1.1.1 A language-independent platform

A language compiler targets the .NET platform by translating source code to IL, as we see in figure 1.1.

The output from compilation consists of IL and metadata. IL can be described as an assembly language for a stack-based, virtual, .NET “CPU.” In this respect, it is similar to the p-code generated by early versions of Visual Basic, or to the bytecode emitted by a Java compiler. However, IL is fully compiled before it is executed. A further difference is that IL was not designed with a particular programming language in mind. Instead, IL statements manipulate common types shared by all .NET languages. This is known as the *Common Type System*, or CTS. A .NET type is more than just a data type; .NET types are typically defined by classes that include both code and data members.

At run time, the *Common Language Runtime* (CLR, in figure 1.1) is responsible for loading and executing a .NET application. To do this, it employs a technique known as *Just-In-Time* (JIT) compilation to translate the IL to native machine code. .NET code is always compiled, never interpreted. So .NET does not use a virtual machine to execute the program. Instead, the IL for each method is JIT-compiled when it is called for the first time. The next time the method is called, the JIT-compiled native code is



**Figure 1.1**  
All languages are  
compiled to IL

executed. (This is the general case, since .NET code can also be pre-JITted at installation time.)

The compilation process produces a Windows executable file in *portable executable* (PE) format. This has two important implications. First, the CLR neither knows, nor cares, what language was used to create the application or component. It just sees IL. Second, in theory, replacing the JIT compiler is all that's necessary to target a new platform. In practice, this will likely happen first for different versions of Windows including Windows CE and future 64-bit versions of Windows.

### 1.1.2 .NET and managed code

.NET applications, running under the scheme shown in figure 1.1, are referred to as *managed applications*. In contrast, non-.NET Windows applications are known as *unmanaged applications*. Microsoft recognizes that managed and unmanaged code will coexist for many years to come and provides a means to allow both types of code to interoperate. Most common will be the need for .NET applications to coexist alongside COM in the immediate future. Therefore, Microsoft has endowed .NET with the ability to work with unmanaged COM components. It is also possible to register a .NET component as a COM object. Similarly, for Win32 API access, .NET allows managed code to call unmanaged functions in a Windows dynamic-link library (DLL). We'll look at some examples of .NET/COM/Win32 interoperation in the following chapters.

In addition to JITting the code, the CLR manages applications by taking responsibility for loading and verifying code, garbage collection, protecting applications from each other, enforcing security, providing debugging and profiling services, and

supporting versioning and deployment. Code management by the CLR provides an extra layer that decouples the application from the operating system. In the past, the services provided by this layer would have been implemented in the application itself, provided by the operating system, or done without.

You may be wondering about the metadata emitted along with IL by the language compilers shown in figure 1.1. This is a key feature of .NET. For those of you familiar with COM or CORBA, the metadata can best be described as a form of Interface Definition Language (IDL) that is automatically produced by the language compiler. Metadata describes types including their fields, properties, method signatures, and supported operations. By producing this data automatically at compile time, .NET components are self-describing and no additional plumbing is required to get .NET components, written in different programming languages, to interoperate seamlessly.

## 1.2 A FIRST .NET PROGRAM

Without further delay, let's take a look at a simple .NET application. The program shown in listing 1.1 is a simple C# command-line program which greets the user.

**Listing 1.1 Hello from C#**

```
// file      : hello.cs
// compile  : csc hello.cs

using System;

class Hello {

    public static void Main() {
        Console.WriteLine("Hello from C#");
    }
}
```

Every C# program must contain at least one class. In this case, that class is `Hello` and its `Main` method is the program's entry point where execution begins. (A member function of a class is known as a *method*.) To display the greeting, the program calls:

```
Console.WriteLine("Hello from C#");
```

This calls the `WriteLine` method of the `Console` class, which is contained in the `System` namespace, to display the message. The `System` namespace is part of .NET's Framework class library. We could have coded this call, as follows:

```
System.Console.WriteLine("Hello from C#");
```

Instead we declared the `System` namespace at the start of our program:

```
using System;
```

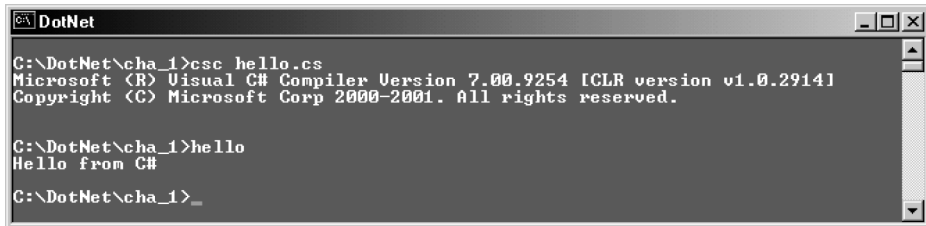
This allows us to omit the namespace name and provides a shorthand for referring to `System` classes within our program.

This short example demonstrates the use of .NET's Framework class library, a huge repository of useful classes, which we can use in our .NET applications. These classes are grouped by function and logically arranged in namespaces. We'll look at some common namespaces in a moment.

### 1.2.1 Compiling the C# Hello program

To compile and test this example, you'll need a copy of the .NET SDK, or Visual Studio .NET. At the time of writing, the SDK could be downloaded from the Microsoft Developer Network site, <http://www.msdn.com>.

To compile and run this program, open a command window, and use the C# command-line compiler, as shown in figure 1.2.



```
C:\DotNet\cha_1>csc hello.cs
Microsoft (R) Visual C# Compiler Version 7.00.9254 [CLR version v1.0.29141
Copyright (C) Microsoft Corp 2000-2001. All rights reserved.

C:\DotNet\cha_1>hello
Hello from C#

C:\DotNet\cha_1>
```

Figure 1.2 Compiling and running the C# Hello program

If *csc.exe* is not found, you'll have to add the directory where it resides to your path. This directory will depend on the version of .NET you are using and should look like *C:\WINNT\Microsoft.NET\Framework\<.NET Version>*.

We'll be using C# for the programming examples in this book. For readers unfamiliar with the language, a complete introduction is provided in appendix A. However, before we commit to C#, let's take a brief look at Visual Basic .NET.

### 1.2.2 A Visual Basic .NET Hello program

For comparison, listing 1.2 shows the same program coded in Visual Basic .NET.

#### Listing 1.2 Hello from VB.NET

```
' file      : hello.vb
' compile  : vbc hello.vb

Imports System

module Hello

    sub main()
        Console.WriteLine("Hello from VB.NET")
    end sub

end module
```

You can see that the Visual Basic .NET version of the program is very similar. Specifically, the Visual Basic .NET program uses the same `Console` class from the `System` namespace. The Framework class library is part of the .NET platform and is not the preserve of a particular programming language.

In general, there is less difference between C# and Visual Basic .NET than you might expect, and those differences that exist are mostly syntax-based. C# and Visual Basic .NET programmers use the same framework classes and deal with the same .NET concepts, including namespaces, classes, the CLR, and so forth. Also, as we saw in figure 1.1, both C# and Visual Basic .NET programs are compiled to IL. If you were to examine the generated IL for the previous C# and Visual Basic .NET examples, you'd find the output almost identical.

### **1.3 THE PLATFORM VS. THE PROGRAMMING LANGUAGE**

IL is not a typical assembly language in the tradition of machine assembly languages such as 8080 or 6809 assembler. Instead, it is comprised of an instruction set and an array of features that are designed to support the essential operations and characteristics of many modern, object-oriented languages. The focus of .NET is on a common object system instead of a particular programming language.

The CLR directly supports many features which might ordinarily be features of the programming language. This includes a language-neutral type system with support for classes, inheritance, polymorphism, dynamic binding, memory management, garbage collection, exception handling, and more. For example, the same garbage collector is responsible for deleting unused objects from the heap and reclaiming memory, no matter which programming language was used to code the application. So, inclusion of features such as these in the CLR provides a common bridge to facilitate language interoperation and component integration.

To facilitate cross-language interoperability, .NET includes a *Common Language Specification*, or CLS, that represents a common standard to which .NET types should adhere. This standard lays down rules relating to allowed primitive types, array bounds, reference types, members, exceptions, attributes, events, delegates, and so forth. Components and libraries which adhere to this standard are said to be CLS-compliant.

Cross-language inheritance presents no special challenge when CLS-compliant code is involved. You can create a base class using Visual Basic .NET, derive a C# class from it, and seamlessly step through both with a source-level debugger. This level of language interoperability is probably one of .NET's greatest strengths. Many powerful and elegant programming languages, commonly available on other platforms, have failed to become first-class citizens of the Windows world due to their limited integration with the platform. .NET promises to change this.

## 1.4 EXPLORING THE .NET FRAMEWORK CLASS LIBRARY

In the early days of Windows development, applications were typically coded in C and interaction with the operating system was through C-based API function calls into system DLLs. This was a natural consequence of the fact that much of Windows itself was written in C. Over the years, the emphasis has gradually shifted to more flexible COM-based interfaces that can be invoked by both traditional C-based applications and by scripting languages.

.NET supplants both these approaches with a new language-independent framework class library. Under the .NET Framework, everything is an object, from a humble C# or Visual Basic .NET array (`System.Array`), to a directory under the file system (`System.IO.Directory`), to the garbage collector itself (`System.GC`).

### 1.4.1 An overview of important namespaces

As we've already noted, the .NET Framework classes are grouped by function and logically organized into namespaces. There are almost 100 namespaces shipped with the .NET SDK, and some contain dozens of classes. Therefore we can't explore them all here. Even if we could, we'd soon forget most of them. So table 1.1 lists some of the more commonly used namespaces and provides a brief description of each.

**Table 1.1 Common .NET namespaces**

Namespace	Functional area of contained classes
Microsoft.CSharp	Compilation and code generation using the C# language
Microsoft.JScript	Compilation and code generation using the JScript .NET language
Microsoft.VisualBasic	Compilation and code generation using the Visual Basic .NET language
Microsoft.Win32	Windows registry access and Windows system events
System	Commonly used value and reference data types, events and event handlers, interfaces, attributes, exceptions, and more. This is the most important namespace
System.Collections	Collection types, such as lists, queues, arrays, hashables, and dictionaries
System.ComponentModel	Run-time and design-time behavior of components
System.Configuration	Access to .NET Framework configuration settings
System.Data	ADO.NET types
System.Data.SqlClient	SQL Server .NET Data Provider types
System.Data.SqlTypes	Native SQL Server data types
System.Diagnostics	Application debugging and tracing. Also Windows Event Log class.

*continued on next page*

**Table 1.1 Common .NET namespaces (continued)**

<b>Namespace</b>	<b>Functional area of contained classes</b>
System.DirectoryServices	Active Directory access using service providers such as LDAP and NDS
System.Drawing	Windows GDI+ graphics types
System.Globalization	Culture-related types embracing language, string sort order, country/region, calendar, date, currency and number formats
System.IO	Reading and writing of streams and files
System.Messaging	Sending, receiving, and managing queued messages (MSMQ)
System.Net	Simple API for network protocols such as DNS and HTTP
System.Net.Sockets	API for TCP/UDP sockets
System.Reflection	Access to loaded types and their members
System.Reflection.Emit	Metadata/IL emission and PE file generation
System.Resources	Creation and management of culture-specific application resources
System.Runtime.InteropServices	Accessing COM objects and native APIs
System.Runtime.Remoting	Creation and configuration of distributed objects
System.Runtime.Remoting.Channels	Managing remoting channels and channel sinks
System.Runtime.Remoting.Channels.Http	HTTP (SOAP) channel management
System.Runtime.Remoting.Channels.Tcp	TCP (binary) channel management
System.Runtime.Remoting.Lifetime	Managing the lifetime of remote objects
System.Security	Accessing the underlying CLR security system
System.Security.Permissions	Controlling access to operations and resources based on policy
System.Security.Policy	Code groups, membership conditions, and evidence, which define the rules applied by the CLR security policy system
System.Security.Principal	Identity/Principal classes, interfaces, and enumerations used in role-based security
System.ServiceProcess	Windows service installation and execution
System.Text	Text encoding and conversion for ASCII, Unicode, UTF-7, and UTF-8
System.Text.RegularExpressions	Access to the built-in regular expression engine
System.Threading	Classes and interfaces for multithreaded programming
System.Timers	Timer component for raising events at specified intervals
System.Web	Browser/server communication including commonly used ASPNET classes such as HttpApplication, HttpRequest, and HttpResponse
System.Web.Configuration	ASPNET configuration classes and enumerations

*continued on next page*

**Table 1.1 Common .NET namespaces (continued)**

Namespace	Functional area of contained classes
System.Web.Services	Building and using Web services
System.Web.Services.Description	Describing a Web service using WSDL
System.Web.Services.Discovery	Discovering Web services via DISCO
System.Web.SessionState	Access to ASPNET session state
System.Web.UI	Creation of ASPNET Web pages and controls
System.Web.UI.Design	Extending design time support for Web Forms
System.Web.UI.Design.WebControls	Extending design time support for Web controls
System.Web.UI.HtmlControls	Creating HTML server controls
System.Web.UI.WebControls	Creating Web server controls
System.Windows.Forms	Creating Windows Forms-based user interfaces and controls
System.Windows.Forms.Design	Extending design-time support for Windows Forms
System.Xml	Standards-based XML support
System.Xml.Schema	Standards-based support for XML schemas
System.Xml.Serialization	Serializing objects into XML documents or streams
System.Xml.XPath	The XPath parser and evaluation engine
System.Xml.Xsl	Support for XSL transformations

The list of namespaces in table 1.1 includes only the more commonly used namespaces, most of which are used in examples in the chapters that follow.

## 1.4.2 Programming with the .NET Framework classes

Namespaces provide a convenient way to logically group related classes together. They also prevent name clashes where two or more classes have the same name, but reside in different namespaces. The classes themselves physically reside in DLL files that are shipped with the .NET Framework. Depending on the version of .NET you are using, these DLLs can be found in the *C:\WINNT\Microsoft.NET\Framework\<.NET Version>* directory.

The most common classes reside in the core library file, *mscorlib.dll*. When you use classes that reside in other DLLs, you must refer to the DLL when you compile your program. For example, the `SecurityIdentity` class from the `System.EnterpriseServices` namespace resides in the *System.Enterpriseservices.dll*. To compile a C# program that uses this class, you need to use the C# compiler's `/reference` option and provide the DLL name:

```
csc /reference:System.Enterpriseservices.dll MyProg.cs
```

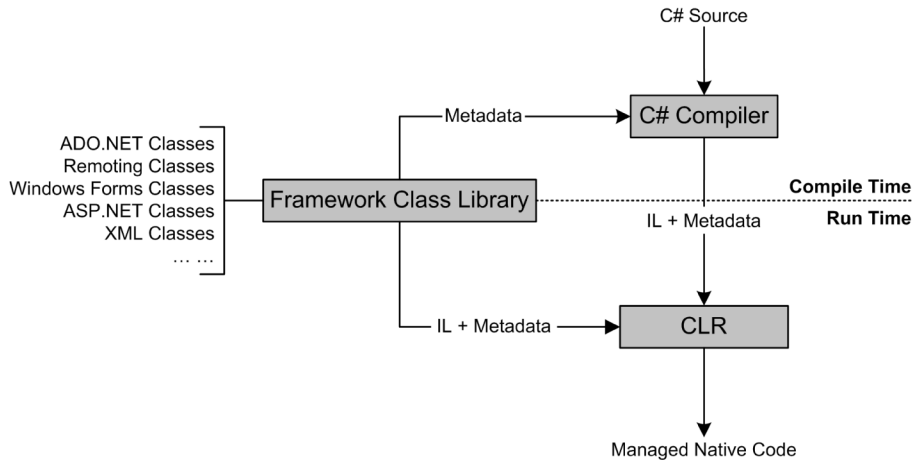
Or, for short:

```
csc /r:System.Enterpriseservices.dll MyProg.cs
```

Note that there isn't a one-to-one correspondence between namespaces and DLLs. A DLL may contain classes from several different namespaces, while classes from the same namespace may be physically distributed among several DLLs.

**NOTE** For each class in the Framework, the .NET reference documentation gives both the containing namespace name, and the name of the physical file where the class resides.

Figure 1.3 illustrates how the Framework fits into the .NET development model.



**Figure 1.3 Architecture of .NET**

Metadata flows from the Framework class library to the C# compiler. The compiler uses the metadata to resolve references to types at compile time. Unlike C and C++, C# does not use header files, nor is there an explicit linkage stage in the build process.

In figure 1.3, we also see the CLR pulling in the IL and metadata for both the application and the Framework classes it uses. This process is analogous to dynamic linking under Windows, but with the extra .NET bells and whistles described earlier, such as verifying type-safety and enforcing version policy.

### 1.4.3 What happened to ASP and ADO?

You may be familiar with Active Server Pages (ASP) and ActiveX Data Objects (ADO), and you may wonder how they fit into the .NET puzzle. In fact, they are implemented as part of the class library. For example, the `System.Data`, `System.Data.SqlClient`, and `System.Data.SqlTypes` namespaces, shown in table 1.1, make up part of the new ADO.NET subsystem. Likewise, the `System.Web`, `System.Web.UI`, and several other namespaces listed in table 1.1, make up part of the new ASP.NET subsystem. In the same way, the Framework also embraces Windows GUI development with the Windows Forms classes in `System.Windows.Forms`.

This is consistent with the .NET approach in which previously disparate and often-unrelated areas of Windows and Web-based development are combined in a single new framework. The historical division of skills between Visual C++ component developers, Visual Basic graphical user interface (GUI) developers, and VBScript/HTML Web developers, is a thing of the past.

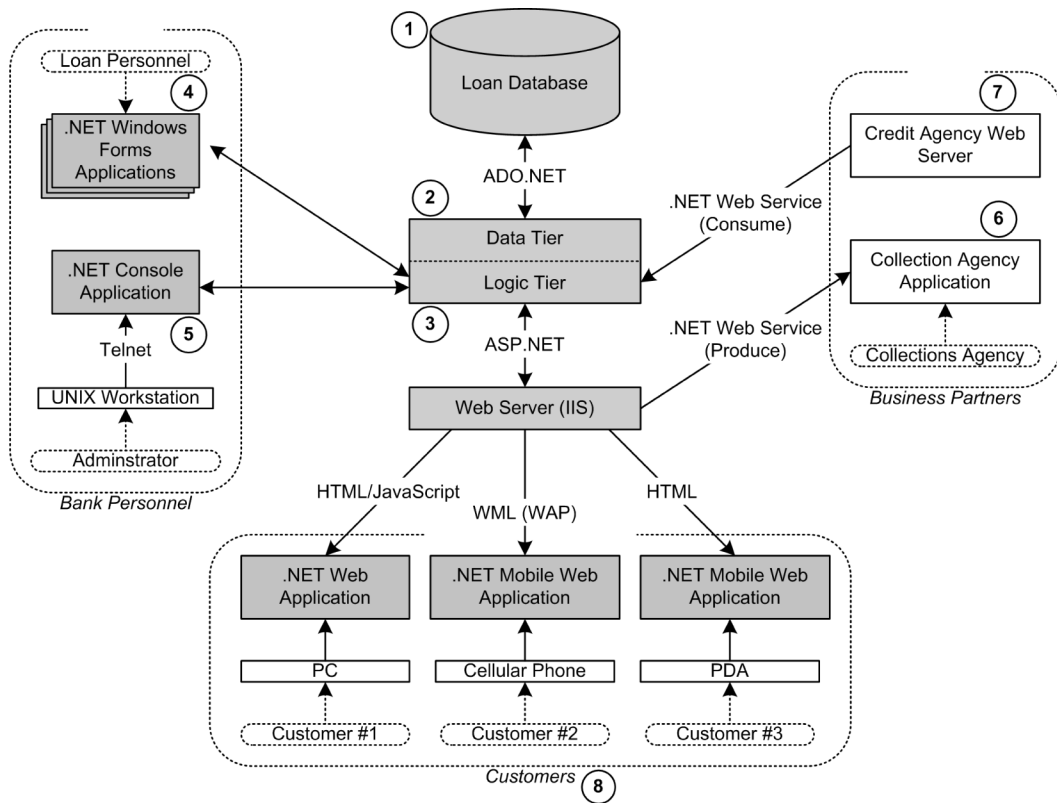
Under .NET, coding Web-based applications is no longer a separate discipline and the artificial divide between component development and scripting no longer exists. For example, you can include C# code in an ASP.NET page. The first time the page is requested, the C# compiler compiles the code and caches it for later use. This means that the same set of programming skills can be employed to develop both Windows and Web-based applications. Likewise, Visual Basic developers have the full power of Visual Basic .NET with which to develop ASP applications, while VBScript has been retired.

## 1.5 **PUTTING .NET TO WORK**

The breadth of coverage of .NET, and the way it unifies previously separate programming disciplines makes it possible to develop and deploy complex distributed applications like never before. Let's consider an example. Figure 1.4 depicts a loan department in a bank together with the applications, and users, involved with the system.

Let's see how we use .NET to develop the applications and components required to build and deploy a loan system like this:

- 1 *Loan database*—The database contains the data for individual loan accounts. It also contains views of the bank's customer database. The loan database might be stored in SQL Server, Oracle, Informix, DB2, or some similar database management system.
- 2 *Data tier*—This is the data tier in an N-tier, client/server arrangement. The data tier uses ADO.NET to talk to the database system and it presents an object-oriented view of the data to the logic tier. In other words, it maps database records and fields to objects that represent customers, loans, payments, and so forth. We could use C#, or Visual Basic .NET, or some other .NET-compliant language to implement the data tier.
- 3 *Logic tier*—The logic tier contains the business rules. We can look upon this layer as the engine at the heart of the system. Once again, we can use C# or Visual Basic .NET, or some other .NET language here. Our choice of programming language is not affected by the language used to develop the data tier. Cross-language compatibility, including cross-language inheritance, means that we can pick the best language for the task at hand. The CLR and the CLS combine to ensure that we can freely mix and match languages, as required.
- 4 *Internal loan department applications*—We would probably choose to develop internal-use applications as traditional Windows GUI applications. Using typical client/server techniques, these internal applications would talk directly to



**Figure 1.4** A sample bank loan system which uses .NET

the logic layer across the bank's local area network. The traditional Windows GUI model is known in .NET as Windows Forms. It is similar to the tried-and-tested Visual Basic forms model. Using Visual Studio .NET, forms can be designed using a drag-and-drop approach. Windows Forms can contain all the familiar Windows controls, such as the buttons, check boxes, labels, and list boxes. It also contains a new version of the Windows Graphical Device Interface (GDI), a new printing framework, a new architecture for controls and containers, and a simple programming model similar to Visual Basic 6.0 and earlier versions of Visual Basic.

- 5 *Administrator console applications*—Perhaps the bank uses Informix on UNIX as the database management system. If so, we may have administrators who wish to run .NET applications from their UNIX workstations. .NET's humble `System.Console` class can be used to create command line applications that operate over a telnet connection. For example, using Visual Basic .NET or C#, we might write a console application to allow an administrator on another

platform to archive expired loans. Console I/O and a modern GUI cannot be compared for usability, but it is useful and appropriate in a case such as this.

- 6 *Business partner Web service (produce)*—Here we have a business partner, a collection agency, hired by the bank to pursue payment of delinquent loans. In the past, we might have designed a batch application to extract delinquent accounts every day and export them to a flat file for transmission to the collection agency. With .NET, we can implement this function as a Web service that exposes delinquent accounts to remote clients. This means that the collection agency can hook its own applications into the bank's loan system and extract delinquent accounts as required. While human surfers consume Web sites, Web services are consumed by other applications. They promise a future, federal model where independent Web services will be produced, consumed, reused, and combined in the creation of powerful, interconnected applications. The simplicity of the Web service model will likely give rise to a rapid increase in automated, business-to-business, e-commerce applications.
- 7 *Business partner Web service (consume)*—The loan system might also be a consumer of Web services produced by other business partners. Here, we see a credit agency that produces a Web service to enable commercial customers to perform credit checks on loan applicants. .NET provides the tools to build a client to consume this service and integrate it into our loan system.
- 8 *Customer Web-based applications*—Using ASP.NET, we can quickly deploy a Web-based loan application system which gives the customer immediate access to the loan system. Also, .NET's Mobile Internet Toolkit means that we can deploy and integrate an interface that allows customers with handheld devices, such as Web-enabled phones and personal digital assistants (PDAs), to access the loan system for balance enquiries and transfers.

Perhaps it's more instructive to consider what's missing from figure 1.4. For example, there are no CORBA or DCOM components. Instead, we leverage the Web server using ASP.NET's XML Web services infrastructure to expose system functions to remote callers.

Neither do we employ any additional servers or filters to support multiple mobile devices such as Web-enabled phones and hand-held PCs. We use the controls from the Mobile Internet Toolkit to take care of detecting and supporting multiple devices.

Although not evident in figure 1.4, no special steps are taken to support multiple browsers. Instead, we use ASP.NET controls that automatically take care of browser compatibility issues. This means that our ASP.NET applications can take advantage of the extended features of higher-level browsers, while automatically producing plain old HTML for older browsers.

This example depicts a moderately complex system involving multiple interfaces and different user types, including bank staff, customers, and business partners, who

are distributed in different locations working on multiple platforms. .NET provides us with everything we need to build and deploy a system such as this.

Starting in chapter 3, we begin a case study containing all the essential features of the system depicted in figure 1.4. We develop a video poker machine and deploy it as a console application, a Windows application, a Web-based application, a remote object, a mobile application, an XML Web service, a Windows service, and a message-based service. We implement a 3-tier client/server architecture, and our poker machine will exhibit all the essential features of a modern, multi-interface, distributed application.

## **1.6 SUMMARY**

We have taken a first look at the features of the .NET platform. We learned about the CLR and that .NET is fundamentally agnostic about the choice of programming language. We explored the Framework class library and we noted that subsystems such as ADO.NET and ASP.NET are integral pieces of the Framework. We also considered the case of a distributed bank loan system and how we might use .NET to build it.

In the next chapter, we explore .NET types and assemblies, the fundamental building blocks of .NET applications.