Table of Contents

Table of Contents

Table of Contents

The book you are holding is aimed at core material and fundamentals. The Introduction will fill you in with that aspect of it.

I encourage you to send me your contributions, comments, and any suggestions on fixing or enhancing the material presented in this book. You can write me at jshapiro@sdamag.com or visit www.sdamag.com. Any contribution you wish to make will be considered and you'll be asked permission to include it in future editions or in the solutions.

I sincerely hope you will enjoy reading this book and gain the enrichment that I have garnered from writing it.

Acknowledgments

Many people made this book possible. Besides editors and production people and writers, authors, testers, and reviewers, a great many people who did not have a direct involvement in this book nevertheless provided contributions which were indispensable. I would like to thank these dear friends first.

During the early stages of this book, I relied heavily on my coworker and assistant, Saby Blanco, who was sadly taken from us without warning in late 2001. (I know God had a reason for recalling Saby in a heartbeat; I just wish I knew what that reason was). I have dedicated this book to his memory and to thank him for his friendship and help. He is sadly missed by many. He was a terrific person.

My wife and dear friend, Kim, has certainly had it rough in recent years, but without her unfaltering commitment and support it would have been very difficult to reach this stage in the life of not only this book but in my other books as well.

A special thank you to my sister, Lesley Kalish, for her assumption of many of my family responsibilities that made it possible for me to dedicate the time I did to this project. I owe the same level of appreciation to my uncle, Charlie Frank, for his support, love, and friendship.

I also owe more that a few words of thanks to my wife's family, the Zagnoevs, and in particular to my father−in−law and mother−in−law, Barney and Entha Zagnoev, whose support in this "venture" and several others in the not−too−distant past, has meant a great deal to me.

Many coworkers and collegues in the past years made it possible for me to put the words and code in these pages between book covers. They include Steven Cohen at TempArt who always happens to call just when you think it's time to give him a shout; Armando Blanco for his support in various technical fields over the years and for his friendship; and Mike Costolo at C&L Insurance, Inc., who deserves a special thank you for his support, especially during the weeks and months this book had me deep in living in an alternate reality.

Two people deserve special thanks for the effort and support they have given me over the past eight yearsespecially with respect to my career. They are Stephen Kain, of the law firm Polatsek and Sclafani, and my book agent, David Fugate, of Waterside Productions, Inc.

No author can boast that he or she did a book single−handedly. And no matter how much effort goes into the creative side, without the help and dedication of editors and production people a good book can very quickly go bad. On that note, I would first like to thank the Production Editor at Osborne, Elizabeth Seymour, for going more than the extra mile for me. Besides the hard work and commitment to the publishing task, her support, understanding, and tolerance (of me) are greatly appreciated.

I also would like to thank my publishers and the all the production and editorial staff that helped keep this

Chapter 1

The first chapter is not so much about Visual Basic .NET as it is about programming in general and programming in .NET in particular. Experienced programmers will likely skim over this chapter, but newcomers would benefit from the background to programming in general, and from finding out what Visual Basic .NET and the .NET Framework have to offer.

The chapter takes you through ages of procedural and structured programming, and into the object−oriented paradigm. We will discuss modularity, class cohesion, and related topics.

The largest section in this chapter goes into what makes a pure object−oriented language. It discusses the so−called "three corners of OO": inheritance, polymorphism, and encapsulation. It also points out how many constructs, like encapsulation, are rooted in programming models, pre−OO. Most important is that you'll see how Visual Basic now fits the bill as an extremely powerful and pure OO language.

The chapter covers the differences between object−oriented programming (OOP) and object−based programming (OBD). There is also a discussion about frameworks.

The concept of patterns in software development is a very important subject. The subject of patterns is introduced in this chapter. Several chapters go into key structural and behavioral patterns in detail; these include Composite, State, Bridge, and Singleton. You will see how many patterns that have been used for years in OO software development lay the foundations for many sophisticated technologies in .NET. Delegates are a case in point.

Chapter 2

It is important to get up to speed with the .NET Framework and the common language runtime (CLR) as soon as possible. While it is true you can install the CLR and forget it for many applications, there are a lot of things you need to be aware of when it comes to how your code is executed. This chapter goes into Microsoft intermediate language, how your application code gets packaged into assemblies, and how the runtime locates and runs your code.

When I first started this book I thought it would not be necessary to go into the CLR in any detail; maybe give the subject a few paragraphs. Then I tried to deploy an application for a client to the production servers, only to discover the code was unable to run due to some obscure security condition. While this chapter presents the basics of security (the runtime environment and the CLR), the information I gained from learning about the CLR made all the difference. A few tweaks here and there, and the code was up and running.

You need to know about the assembly cache, side−by−side execution, the Common Language Specification, and .NET security. While you do not need to become a guru on all the subject matter covered in Chapter 2, you'll have the confidence to move your software off your development workstation and know what it needs to run with in the world at large.

Chapter 3

This chapter aims at making you productive with Visual Studio .NET as quickly as possible. The chapter has been designed to have you learning important points from the get−go, so that you'll be able to have code compiled and running before you reach the end of the chapter.

Chapter 17

The last chapter deals with debugging and tracing. It can be argued that this chapter should have come much sooner, but I believe that if you are just starting out programming, using the debugging tools and facilities is not going to be easy. After all, using the **Debug** class, performance counters, trace listeners, the **Trace** class, and numerous other complex classes in the **System.Diagnostics** namespace is not straightforward without a basic understanding of the core language. Once you are up to speed with the concepts presented in the earlier chapters you'll find Chapter 17 a cinch to read, and the debugging aids like breakpoints and code step−through features and the classes, a matter of course.

Conventions

Many of the conventions used in this book are self−evident. However, I have added a number of symbols in many tables that differentiate between properties, methods, fields, static methods, and instance methods. These symbols are listed in the following table.

Symbol	Explanation
(a)	Denotes an abstract class or method
(d)	Denotes a **Delegate** object
(fi)	Denotes a field
(fl)	Denotes final
(i)	Denotes an instance method
(m)	Denotes a method
(p)	Denotes a property
(s)	Denotes a static method

A number of tables provide lists of class members. These tables will give you an idea of what constructs are available to the class and may or may not correspond to an element further explained in the text. However, in most of the cases the tables are abridged. They especially do not list the members that are always inherited from the root **Object**. Defer to the .NET SDK for the full picture.

Teaming

Structured programming allows us to build a software product in teams. As with the Avenger, various programmers, grouped into designers and implementers, can work independently on different parts of a product. Structured programming also lets us delegate according to skill sets, which include writing the stored procedures; driving the code on the web site; sorting the data structures; and controlling specifications, class diagrams, use cases, and documentation.

Structural Nada?

Despite the progress made since the advent of structure−oriented programming, many designers still do not program in a "structured way." They program without regard for the interrelationships of modules, organization for reuse, overall structure, or protection of data. Software project managers often talk in terms of the "number of function points" in an application or system, so as to express the extent or size of the project. But talking about so−called function points is meaningless if there is no way to express the architecture of that system, its components, their interrelationships, and its various structures.

As I said earlier, many courses in OOP first teach structured programming, because a good OO program is also a well−structured one. Many first−year programming courses that begin with C++ or Java require you to learn structured programming in these languages before you learn to "objectify" your code.

Object−Based Programming

As structured programming methods became more refined, it became apparent to the engineers that modules of code could be related to as objects. Many languages gradually became more object−based, especially as compilers emerged that could enforce encapsulation and cater to the compiling, linking, and binding of numerous modules into complete applications.

A good example of object−based programming plots the transformation of C into C++. Today C++ is what you would call a *hybrid* language because it allows you to code standard C modules, to add objects and do object−based programming, or to invoke the object−oriented features of the language such as inheritance and polymorphism.

Visual Basic is an excellent example of the evolution of a language. It was born in May 1991, the child of a shapeless language parent, BASIC. Then it grew into its structured programming and module−based programming stages (all the way to VB 6). Now it has matured into a true object−oriented language, Visual Basic .NET. Today the language has pure, compiler−enforced support for inheritance, polymorphism, and all the great features that pure OO languages support.

Object−Oriented Software Development

Computer specifications are advancing at a record pace while prices for these systems are falling rapidly. As computer hardware gets cheaper, software is getting more and more expensive. In order to create machines that will address bigger and more complex problems, we need software that is more complex and thus more expensive. There is also new hardware such as mobile devices, super fast processors, cheap memory, and tiny hard disks that require sophisticated new software. The problem is further compounded by our desire for ever−faster and more powerful computers.

Chapter 2: Visual Basic .NET and the .NET Framework

Overview

Chapter 1 introduced you to the .NET Framework and hinted at what's possible in the .NET runtime environment.

This chapter focuses on the common language runtime, the CLR. In tackling this subject we will be able to design and code applications with the runtime in mind; in particular, the issue of memory management represents the biggest change in the way we write applications. Knowing about the runtime is crucial for programming with the correct security model, implementing exception handling, referencing the correct assemblies to target namespaces, debugging assemblies, and otherwise managing assemblies (deployment and maintenance).

Acquiring background on how the runtime operates and executes your code will allow you to become fully proficient in .NET programming. It's admirable being an expert in software design and constructionand this book is mainly about thatbut the best−written applications are useless if they get "trampled" in the runtime environment.

However, we don't need to cover everything about the CLR. We will focus primarily on the concerns of Visual Basic developersdeploying assemblies, programming for security, and performanceand less on the needs of framework developerswriting their own .NET runtimes, compilers, and languages. You'll want to closely examine the discussion on assemblies and intermediate−language (IL) code, because in later chapters we will evaluate the IL with respect to performance and debugging issues.

This chapter deals with theory. Here we examine the following key components of the .NET Framework:

- **The common type system (CTS)** This system provides the type architecture of the framework and guarantees type safety.
- **The Common Language Specification (CLS)** The specification that all .NET language adopters and compiler−makers employ so that their languages integrate seamlessly into the .NET Framework.
- **The common language runtime (CLR)** The runtime and managed−execution environment in which all .NET applications are allowed to process.

We will then break down the common language runtime into several components to be discussed as follows:

- **Managed execution** We define it and discuss how it differs from other execution environments, such as VBRUN, Smalltalk's runtime, and the Java Virtual Machine. We also introduce the garbage collector.
- **Runtime environment** We discuss how the CLR uses metadata and Microsoft intermediate language (MSIL) to execute code. We also investigate the just−in−time compilation architecture and look briefly at the relevance of application domains vis−à−vis your deployment requirements.
- **Understanding Assemblies** We delve deeply into assemblies, examining how .NET applications, class libraries, and components are packaged. We also touch on the subject of attributesa facility for increasing the programmer's control over the execution and management of code in the runtime environment.
- **The .NET Security Model** We introduce the security architecture of the CLR and how it affects your code and your ability to deploy.

Chapters 9 and 10.

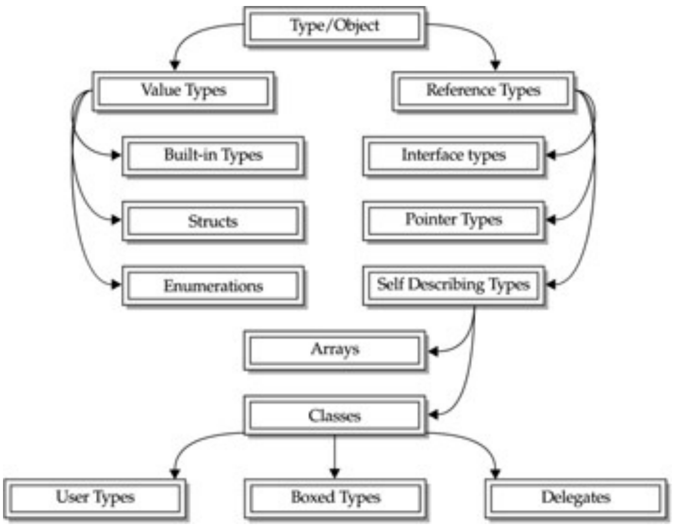


Figure 2−1: The CTS type model, which is the basis for object model and hierarchy

The Common Language Specification

Language interoperability, or *interop*, is considered to be one of the Holy Grails of software developmentand the .NET Framework has risen to the challenge admirably. Opinions about the Common Language Specification (CLS) vary. One that is accurate, but not intended to be complimentary, calls .NET "many languages for one platform" (while Java is "one language for many platforms"). For now it may be true that Windows is the only operating system, but for the programmers of the many languages that support .NET, the CLS is a major breakthrough.

By writing "CLS−compliant code," you construct classes and components that can be used by any language and its respective IDEs and development tools, without the need for complex COM and ActiveX interfaces and registration details. To achieve the magic, the CLS requires that class and component providers expose to consumers only the features that are common to all the .NET languages.

The CLS is really a subset of the common type system, as mentioned earlier. All the rules specified by the common type system in the runtime environment, such as type safety, determine how the CLS governs compliance at the code−construction and compilation levels. The CTS protects the integrity of code by ensuring *type safety*; code constructs that risk type safety are excluded from the CLS. As long as you produce CLS−compliant code, it will be verified by the CTS.

The cliché that says rules are made to be broken is likely to be echoed in various far−flung shops. When you program against the specs in the CLS, you ensure language interoperability for your intended audience and for others. CLS compliance warrants that third parties can rely on your code and that the facilities you want exposed are available to the entire spectrum of developers.

Table 2−1 provides an abridged list of software−development features that must meet CLS compliance rules and indicates whether the feature applies to both developers and compilers (All) or only to compilers.

Table 2−1: Abridged Version of the CLS

Feature	Applies to	What Must Be CLS Compliant

- The parent of the type, or what it inherits from
- Type membership (methods, fields, properties, events)
- Attributes, which are additional elements used on types and their members at runtime

All this data is embedded in the metadata, which allows the assembly contents to be self−describing to the CLR: This eliminates the hassles of application registration, type library registration, and the Interface Definition Language (IDL) required for ActiveX components.

In addition, self−describing containers of code do not need to be identified or registered with the operating system. By packaging metadata within the executable file itself, the assembly is able to describe itself to the CLR as soon as you try to execute it. (The idea is similar to that of carrying a magnetic card with all your personal information, instead of having to access it from an unwieldy database.) This is known as just−in−time execution. You click or launch the application and it immediately tells the CLR, "Here I am, this is what I need, I have permission to look in this directory, I want to call a certain method, I need this much RAM . . .". This may sound like a slow and cumbersome process, but as you will later see, it's not.

Assemblies and their metadata are better for security. You can trust self−describing components more implicitly than you can a file that publicizes itself in the registry these entries date rapidly and their integrity can be easily compromised; they and their implementation counterparts (the DLLs and executables installed on the system) can also become easily separated.

Executable Code

Assemblies do not have *carte blanche* within the CLR. Code is not always passed directly to the JIT compiler. First, the IL code may undergo a thorough inspection if deemed necessary by the platform administrator. The code is given a verification test that is carried out according to the wishes of the network administrator, who might have specified that all .NET code on the machine must be executed according to a certain security policy. The IL code is also checked to make sure nothing malicious has been included. See the section "The .NET Security Model" later in this chapter for details how these checks are carried out. As always, it is crucial to be aware of all this when you get deployment.

Note MSIL is first converted to CPU−specific code by a just−in−time (JIT) compiler specific to a flavor of the Windows operating system. The same set of MSIL, however, can be JIT−compiled and executed on any supported architecture.

The code is also checked to determine that it is type safe, that it doesn't try to access restricted memory locations, and that it references correctly. Objects have to meet stringent safety checks to ensure that they are properly isolated from one another and do not access each other's data. In short, if the verification process discovers that the IL code is not what it claims to be, it is terminated and security exceptions are thrown.

Managed Execution

The .NET just−in−time compiler has been engineered to conserve both memory and resources while giving maximum throughput. Via the code inspection process and self−learning, it can determine what code needs to be compiled immediately and when the rest will be needed. This function is known as JIT compilationthe code is compiled as soon as we need it. When we need machine code "yesterday," we can force compile it and have it ready for action.

Furthermore, the JIT compiler and the CLR manage resources and process "bandwidth" such that tests show CLR code has the potential of running even faster on the managed heap than it does on the unmanaged heap.

For global access to your files, you only need to drop them into the Global Assembly Cache (see the next section). Each computer that carries the CLR is endowed with a GAC (pronounced like *whack*). This "repository" for assemblies is a machine−wide code cache that stores assemblies that have been designated for sharing by more than one application on the machine.

Note The GAC is usually created in the root of your operating system folders. For example, on Windows .NET Server this might be **C:\Winnt\assembly**.

The purpose of the GAC is to expose the assemblies placed in itto applications and services that depend on them. When the CLR needs the assembly required by the application, it will go to the GAC.

Note COM interop code does not have to be installed in the GAC.

If assemblies do not need to be shared among applications, you should store them with their "friends" in private locations. Administrators can then protect the folders if need be, and some of them can be placed entirely off limits to anything but the assemblies that depend on them.

You can use the Windows Installer, or any other .NET−compliant commercial installer, to deploy into the GAC or private folders. The .NET SDK also provides a utility called the Global Assembly Cache tool (GACUTIL.EXE), which you can use for inserting into the cache.

Note Assemblies placed in the GAC must have strong names. See the section on security later in this chapter.

When you are ready to deploy assemblies for ASP.NET applications simply XCOPY or FTP them to the server. When you allow Windows forms or Web service assemblies to be downloaded, they can be packed as either DLL files or compressed .CAB files. You can simply hook up the source via FTP or HTTP and allow the client to download the file through the link.

The benefit of using the Windows Installer, which generates .MSI packages, is you can integrate .NET installation with the Add/Remove Programs option in the Control Panel. You can accomplish installation, removal, and repair in this way.

Understanding Assemblies

The assembly is a "physical" container for at least one built (compiled) executable or class file, module, component, or icon. If the assembly is a library, then the class or classes it harbors are referenced by the fully qualified namespace described in Chapter 4. You still need to reference the assembly in the IDE to gain access to the namespace, so the two are connected at the hip.

If the assembly is an executable file, an application, you reference it by the name of the physical file, which needs an entry point to allow the operating system to initiate its execution. In the next chapter we will create a small application called "Welcome" to demonstrate this. The **welcome.exe** file we produce in that demo is the assembly.

Note Assembly names and namespace names should not be confused. The two are often similar and sometimes identical, but have very little to do with each other, aside from their shared need to "register" an assembly so that Visual Studio can find its way to the namespace.

At the physical level, an assembly is many things, and the organization of its contentsMicrosoft Intermediate fundamental types, and namespaces. The right is called the Members Pane and shows you the class members, as well as enumerated items, variables, and constants. Both panes are illustrated in Figure 3−3.

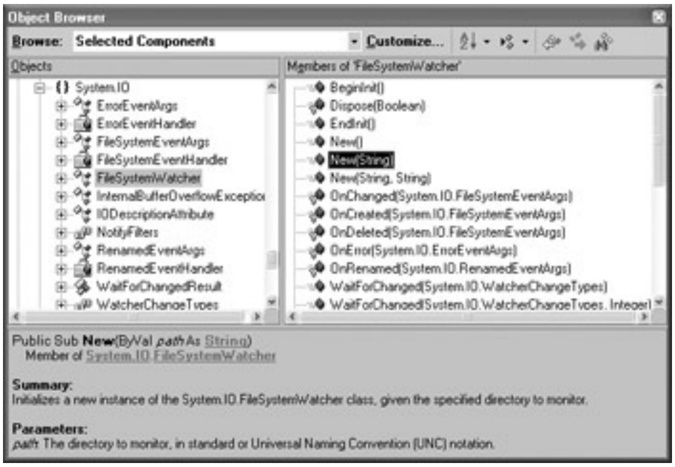


Figure 3−3: The Object Browser

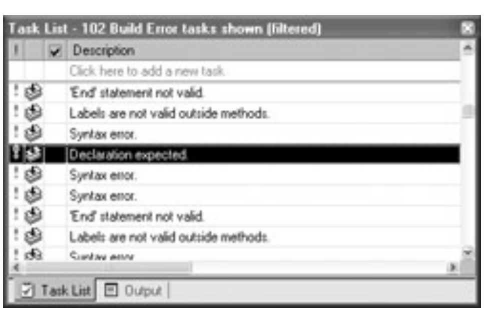
The Description Pane, a third anchor pane in the Object Browser, provides details and further information about the selected class or members. It describes the various components and even displays the method signatures for you. Depending on the class, it may offer examples of the syntax you will use for certain members, including any dependencies, variables, and additional help description that may have been compiled with the object.

Note Object Browser opens as a default tab; however, I recommend you convert it into a dockable window and activate Auto Hide, since you will rely on it constantly.

A drop−down combo−box (Browse) at the top of the Object Browser lets you filter the classes and members pertaining to the objects in your project. You can also customize the browser with buttons that allow you to add additional components to the Object Browser's toolbar. We will return to this in more detail in Chapter 8 and in a number of other chapters.

Task List

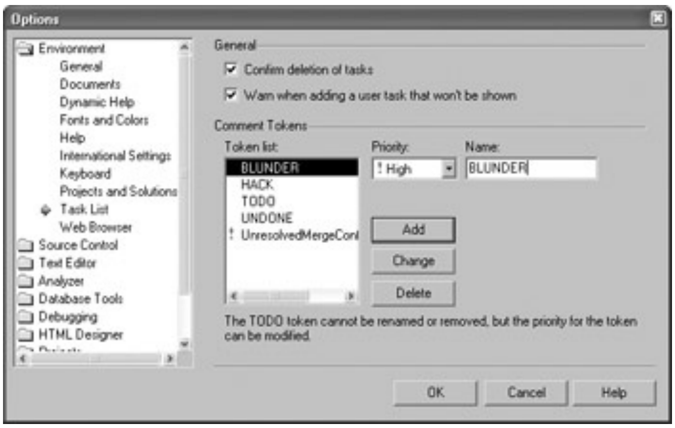
Once you become productive, the Task List will be another indispensable tool. This utility helps you manage tasks within the solution; but most importantly, it displays the compile−time errors and their causes. (See the illustration.)



The class units can contain several predefined tokens, which you can link to from the Task List. These include **TODO**, **UPGRADE_TODO**, and **UPGRADE_WARNING**. As you can see, these tokens have a lot to do with *trying* to migrate classic Visual Basic code.

You can access them by entering the name of your chosen token after the comment symbol (the single quote). The task is automatically added to the Task List. When you need to access the task again, just double−click the item and the appropriate section of code is brought up in the target unit, which moves to the front of all the tabbed documents. Connecting to the errors in your code works the same way. Simply double−click the Task List item and the IDE brings the error to the foreground.

To add your own tokens, a truly terrific feature, go to the Tool menu and select Options. You can then choose the Task List option from the Environment folder.



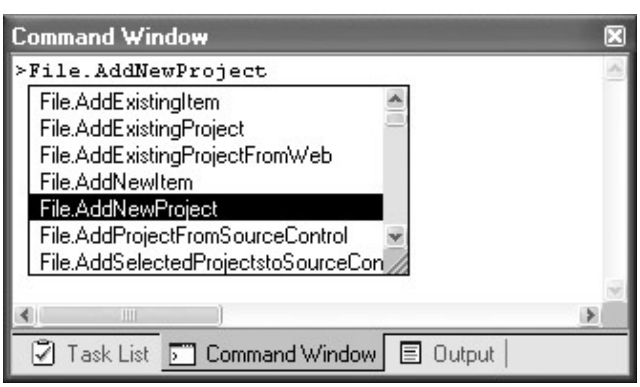
The default only shows build errors, but if you right−click on the Task List you will receive a Show Tasks option that can present the following views: All, Comment, Build Errors, User, Shortcut, Modeling, Policy, Current File, Checked, and Unchecked.

You might be tempted to choose "All," but if there are a lot of comments and errors in the code, the list tends to explode.

Note The Options dialog box can be accessed from the Tools, Options menu. It's worthwhile investigating the Options dialog box because we'll be returning to it in a number of future chapters.

Command Window

The Command Window is another imperative feature of the new Visual Studio .NET IDE. The Command Window has two views, **Command** and **Immediate**.



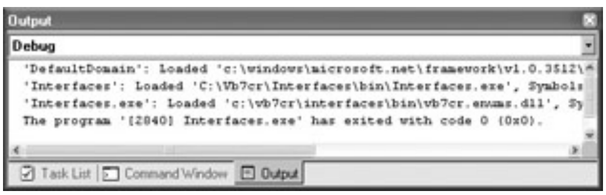
- **Command Mode** Lets you execute Visual Studio Commands without using the IDE menu system. (The illustration shows execution of the **File.AddNewProject** command, which is the command behind the File menu item New, Project.)

- **Immediate Mode** Used for debugging, expression evaluation, and variable modification. If you are familiar with Visual Studio 6, then you'll recognize that this window includes the same functionality as the VS 6 Immediate debugger window.

Both views in the Command Window support Intellisense and Autocomplete. Chapter 17 addresses the use of Immediate Mode for debugging.

Output Window

The Output Window displays build/compiler or diagnostic information depending on the mode it is in. During a build of a project or solution, the window is used to communicate build and compile information. In Debug mode, during processing, the Output Window displays libraries loaded, return codes, and various details being emitted from running code. For example, a special diagnostics **Debug** classdiscussed later in this chapter and in depth in Chapter 17lets you write debug information to the Output Window, shown here.



A third mode this window can switch to, Visio UML, kicks in as soon as you reverse−engineer classes to the UML for loading into Visio (Visio for Enterprise Architects). Selecting Visio UML, Reverse Engineer from the Project menu will achieve this.

Find Results

The Find Results windows (primary and secondary ones) display the results for "search and rescue" operations launched from the sophisticated Find and Replace dialog box. Find and Replace is accessed from the Edit, Find and Replace menu option. From here you can search for tokens, symbols, character strings with standard pattern−matching, regular expressions, and wildcards.

The results of your searches are displayed in the two Find Results dialog boxes and you can search in various places for your targetopen documents, projects, and folders.

Dynamic Help and Search

Dynamic Help is one of the most useful features of this IDE (see Figure 3−1). Simply place your cursor on an element of your code (such as a class name or a method) and Dynamic Help finds and displays a link to the resource in the Visual Studio help system.

Another important feature is the help system's Search facility, which comes equipped with a Help Filter. It will save both time and resources, filtering your help material to just Visual Basic and related information.

Starting from the Start Page

When you first start Visual Studio .NET, the IDE loads up the Start Page, which is the built−in browser's "home page." This HTML page sports a menu of links on the left that provides several options you can choose to "surf to," such as updates and news from Microsoft. Figure 3−1 shows the IDE at its starting position, without any solutions to load. This is how the Start Page should look after a fresh installation.

Loading the Vb7cr Solution

To work with the examples in this book, load the demo solution called Vb7cr from www.osborne.com or from www.sdamag.com. You can follow along with this and try out the code examples in the later chapters. By the time you complete these chapters, you'll have seen a lot of code compiled and will gradually incorporate some of the more advanced features of the IDE, while learning "on−the−job."

As explained in the introduction, we opted not to publish a CD because there are so many paths to downloading the demo projects included with this book. Also, the file is rather small without the assemblies; you will produce them when you build the solution for the first time. You don't need to create a folder for the demo code. Simply unzip it into the root folder or drive of your choice. You can then open the solution into Visual Studio from the File, Open, Project menu items. Browse for the file where you unzipped everything (the root of Vb7cr) and click Open.

Once the solution is loaded, the Solution Explorer comes to life and you can access and open existing projects or create new ones. You can also review its configuration by right−clicking on the solution name and selecting Properties. (See Figure 3−6.)

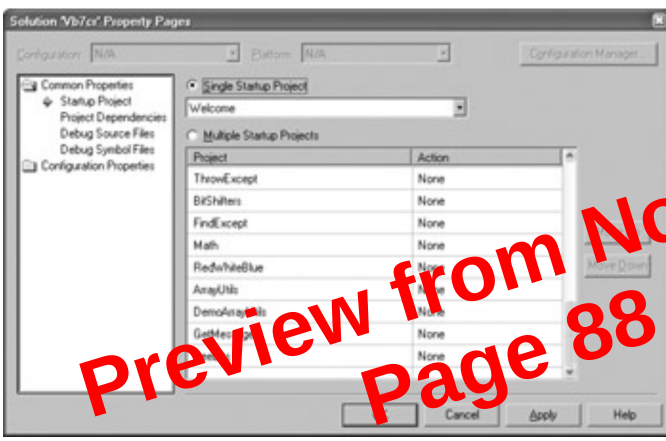
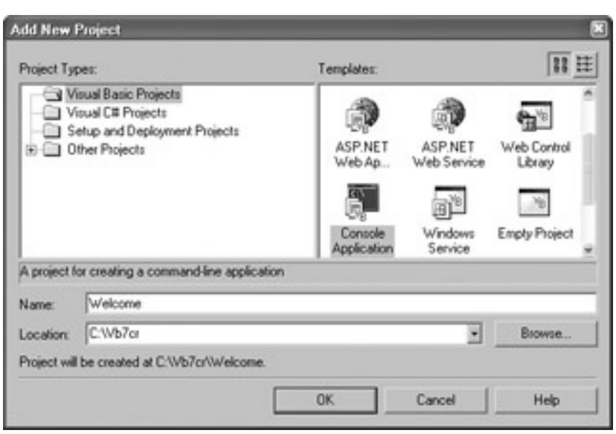


Figure 3−6: The Solutions Properties dialog box

Creating a New Project

When you are ready to add a project to the demo solution, right−click on it in the Solution Explorer and select Add, New Project. (You can perform the same step from the menu option that hangs off the File menu, but this is less expedient.) The dialog box in Figure 3−7 pops up and lets you name your project and specify its location.



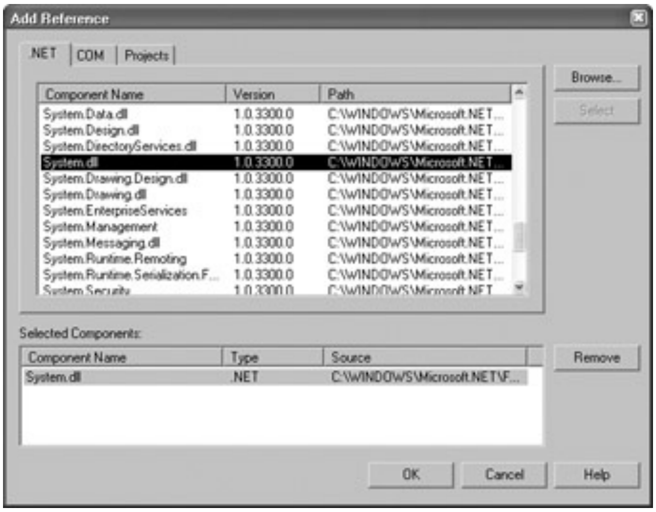


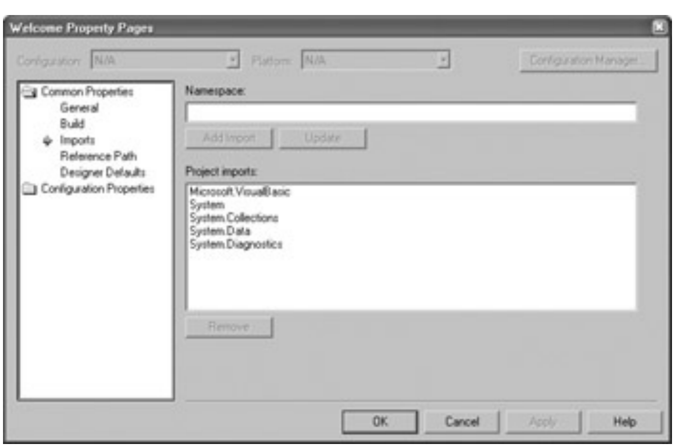
Figure 3−8: The Add Reference dialog box

The first tab lists all the assemblies provided in the Global Assembly Cache, including all of those within the framework libraries. Search for System.DLL in the list and double−click this file, which will be placed in the Selected Components. If it appears in this list, click OK. The System.DLL assembly will now be present in the References section of your project in Solution Explorer, and the error message in your code will vanish.

The good news is this: whenever you create a new project, Visual Studio automatically adds the System, System.Data, and System.XML assemblies to the References section because they contain the base classes you'll need for a minimal application. (These are not bolded here because they are assemblies [DLL files], not classes or namespaces.)

This is important to note since it's not always the case. All custom assemblies and many others that ship with the .NET Framework must first be accessed in this way before you can reference the namespaces and types in your code. Only after your project sees the DLL file containing the namespaces will you be able to reference the namespaces and the object you seek. Don't confuse the process of referencing the assembly with that of denoting the namespace, as shown in the next section.

Looking at your code, you are probably wondering how Visual Studio knows where and what namespaces to select since it's the assemblies that are referenced. It's easy to find out. Right−click on your project and select the Properties options. The Properties dialog box loads. Drill into the Common Properties, Imports folder as illustrated.



You can also select Properties from the Project menu. Go to the Imports sections and add your references there. Word of warningthis box does not check the correctness of your reference so make sure to type the

Properties	Chapters 7, 9
Interface method implementation	Chapter 10
Events and event handling	Chapter 14
Inner or composite types	Chapters 9, 13, 14

Statements and Statement Blocks

The Visual Basic .NET language contains a number of statement constructs used in the construction of code, many of which have been inherited from the earlier versions of VB, such as **With** statement blocks.

The statement block is a statement that encapsulates code within the class itself, to a local scope, such as **IfThenElse** constructs and so on. Table 4−7 leads you to various discussions of statement and statement blocks.

Table 4−7: Statements and Statement Blocks

Statements and Statement Blocks	Principally Discussed in:
Standard block	Chapter 4
Local declaration statements	Chapter 4
With statement blocks	Chapter 4
SyncLock statement blocks	Chapter 7, 16
Event statements and event handler statements	Chapter 14
RaiseEvent statements	Chapter 14
Interface method implementation	Chaptesr 10, 12, 13, 14
Assignment statements	Chapter 4
Compound assignment	Chapters 4, 5
Invocation statements	Chapter 7
Conditional statement blocks	Chapter 6
Loop and iteration statement blocks	Chapter 6
Control−flow statements	Chapter 6
Structured exception handling (SEH) statement blocks	Chapter 11
Unstructured exception handling statement blocks	Chapter 11
Array handling statements	Chapter 12

Expressions

Expressions are sequences of operators and operands that specify a computation and return a value. Table 4−8 directs you toward information on expressions.

Table 4−8: Expressions

Expressions	Principally Discussed in:
Constant expressions	Chapter 4
Variable expressions	Chapter 4
Simple expressions	Chapter 4 and throughout

Literal expressions	Chapter 4
Parenthesized expressions	Chapter 4
New expressions	Chapters 4, 9
Cast and convert expressions	Chapter 4
Me expressions	Chapter 4
GetType expressions	Chapter 9
Is expressions	Chapters 4, 9
Invocation expressions	Chapter 7
Argument and parameter expressions	Chapter 7
Delegate expressions	Chapter 14
Event expressions	Chapter 14
Dictionary expressions	Chapters 12
Index expressions	Chapters 12

Typically, an expression always evaluates to a value at run time, but the compiler may warn you that an expression you are trying to write does not constitute a legitimate expression. For example, the following code does produce a valuable expression:

```
Dim iAm As Boolean = True
Dim jDoo As Integer
Public Shadows Sub ShadowMethod(ByVal argS As String)
  Convert.ToInt32(iAm) = jDoo 'no good, cannot assign right to left
  jDoo = Convert.ToInt32(iAm) 'better
End Sub
```

Operators

Operators are covered everywhere (see Table 4−9), even in this chapter, but each operator is fully defined, and presented with examples of recommended usage and application, in Chapter 5.

Table 4−9: Operators

Operators	Principally Discussed in:
Unary operators	Chapter 5
Logical operators	Chapter 5
Arithmetic operators	Chapter 5
Mod operator	Chapter 5
Exponentiation operators	Chapter 5
Relational operators	Chapter 5
Logical operators	Chapter 5
Is operator	Chapters 8, 9
Like operator	Chapters 7, 8, 9
TypeOf, **TypeOf . . . Is** operators	Chapters 7, 8, 9

The compiler does not consider the type character to be part of the identifier. Also, white space between an identifier and its type character will choke the compiler. Here are some examples using type characters (as you can see, the code is prone to bugs):

Trying to append a type character to an identifier that does not have a type will generate errors.

For example, this declaration will not fly because a standard module cannot be typed:

```
Module Module1# 'try declare a module of type Double
End Module
```

The following code generates type compatibility problems because you cannot assign a value of type **String** to a value of type **Single:**

```
Public Sub TestTypeCharacter()
  Dim mySingle!
  Dim myString$
  mySingle = myString
  Debug.WriteLine(mySingle)
End Sub
```

However, the following fix makes it better:

```
Public Sub TestTypeCharacter()
  Dim mySingle!
  Dim myString$
  mySingle = Convert.ToInt16(myString)
  Debug.WriteLine(mySingle)
End Sub
```


because the **Convert** function converts the **String** value to a **Single** value.

Statements and Blocks

Statements are organized segments of code, which can be organized into blocks. Blocks are made up of labeled lines, and each labeled line begins with an optional label declaration, followed by zero or more statements, and then delimited by colons. For example:

```
Sub-Total:
```

Labels have their own declaration space and do not interfere with other identifiers. In the following example, the type characters for **bar** are used both as the parameter name and as a label name:

```
Function Foo(ByVal bar As Integer) As Integer
   If bar >= 0 Then
      GoTo bar
   End If
   bar = -bar
bar: Return bar
End Function
```

Note: The treatment of labeled blocks is not covered in any meaningful way in this book as is the use of **GoTo** and **On Error** control-flow constructs. Such code is both controversial and outdated. See Chapter 6 for more information.

The first line cannot possibly write the value of **myValue** to the output window because the variable has not yet been declared. It's not difficult to remember this rule; just think of the classic chicken−and−egg or horse−and−cart clichés. In general, all variables and constants at the class level should be declared at the top of the class, and all variables and constants in methods should be declared at the top of the method, just after the signature. It is also important to remember that parameter declarations are scoped to the method and thus their scope is no different to variables or constants declared within the method body (see Chapter 7).

Span

The distance between a declare in a class or a method and the code that references the data is often referred to as *span*. In the following example, the space between the **lastName** declare and the line of code that accesses it is three lines. Thus, we can say that the *span* is three lines.

```
Dim lastName As String
Dim firstName As String
Dim birthDate As Date
GetName(lastName)
```

You can compute the average span in a class to test for its readability. But why should you be concerned about span? The short answer is that it makes it easier to construct code and to read the code you construct. Declares that are not used until much later in a method, or class, force you to keep referring back to areas higher up in the unit to refer to the data in the field.

Note You can declare variables without providing the initial value because the compiler will always provide the default initialization value. For example, if you declare an **Integer** without an initial value, the compiler will automatically assign it 0.

Keeping Lifetimes Short

Keeping lifetimes short also helps to make code less buggy and easier to maintain. Variables that are "live" from the moment a class is instantiated to its death introduce more opportunities for bugs. The live variables also make the code harder to maintain, even if the data is encapsulated in private fields. You are forced to consider all class members and code as possibly misusing a variable, as opposed to localizing the access to one line, a few lines away from where it first declared, or inside the methods that use them (see Chapter 7 for more on method parameter lists, passing arguments and so on).

This is, however, a somewhat controversial subject, because one school of thought says that declaring class variables and constants is better than having to pass the data though methods like a game of rollerball. I personally prefer to keep the class−level variables to a minimum, and instead pass arguments to method fields. The fields are more hidden, more secure, and easier to maintain, and the code is easier to read. In short, this keeps the problem of "hard coding" to a minimum.

Nevertheless, if data needs to be live for the duration the instance is live, then instance data is perfectly reasonable. However, a good rule is to use read−only fields instead of properties where the value is a global constant. This pattern is illustrated in the following code example:

```
Public Structure Int32
  Public Const MaxValue As Integer = 2147483647
  Public Const MinValue As Integer = −2147483648
End Structure
```

Strict and **Option Explicit** directives to the **On** or **Off** position. The **Options** space must precede all other declarations and code. The third space is the Namespace declaration space. Here we see how namespaces and classes are referenced such that they can be accessed from within the implementation space of the class.

The next three chapters deal more specifically with class implementation. Chapter 5 extensively covers the use of operators; Chapter 6 covers flow and control constructs as well as conditional constructs; and Chapter 7 provides the means of accessing the functionality through the construction and provision of the methods of our classes.

Unary Operators

There are three unary operators supported by Visual Basic: **+**, **−**, and **Not** (**Unary Plus**, **Unary Minus**, and **Unary Logical Not**, respectively). They are defined as follows:

- **Unary Plus** The value of the operand
- **Unary Minus** The value of the operand subtracted from zero
- **Unary Logical Not** Performs logical negation on its operand. (This operator also performs bitwise operations on **Byte**, **Short**, **Integer,** and **Long** [and all enumerated types], which we'll discuss later in this chapter.)

Unary Plus is also the additive operator. However, the operator is "overloaded" to perform a concatenation function if the operands are discovered to be of type *string*. For example, the following code

```
Dim S As String = "what is "
Debug.WriteLine(S + "this")
```

writes "what is this" to the Debug window.

Tip Use the **&** symbol for concatenation because it makes code easier to read.

Unary Minus converts a positive number into a negative one. For instance, this simple math

```
x = 3 + −1
Debug.WriteLine(x)
```

writes 2 to the Debug window. However, it's the same thing as 3 minus 1.

The **Unary Logical Not** is different altogether. It can change a **Boolean** result (**False** becomes **True** or **True** becomes **False**). As mentioned earlier, it can also perform a bitwise comparison on a numeric expression. Here are the rules for the **Unary Not Boolean** expressions:

- If the *Expression* is **False,** then the *Result* is **True**
- If the *Expression* is **True**, then the *Result* is **False**

Here's an example:

```
Dim x As Integer = 5
Dim y As Integer = 1
Dim z As Boolean = True
If Not (x > y = z) Then
  Debug.WriteLine("True")
Else
  Debug.WriteLine("False")
End If
```

Normally, the result would be "True" to the debug window, but in the above case truth is **Not** true. The **Boolean** condition inside the parentheses (this entire expression is the operand) is reversedin this case **True** is made **False**. See Chapter 6 for examples of using the **Not** operator in conditional statements, especially **Null If** conditionals. You will also learn about Logical Operators and Bitwise Operators later in this chapter.

If cat loves dog	**Xor dog loves cat**	**is love in the air?**
True	True	False
True	False	True
False	True	True
False	False	False
If cat loves dog	**AndAlso dog loves cat**	**is love in the air?**
True	True	True
True	False	False
False	Irrelevant	False
If cat loves dog	**OrElse dog loves cat**	**is love in the air?**
True	Irrelevant	True
False	True	True
False	False	False

Short−Circuit Logical Operators

The **AndAlso** and **OrElse** are new short−circuit operators introduced to Visual Basic .NET. If you use **And**, the runtime will evaluate the entire expression, even if the first operand is **False**. Compare this to the **And** example in the preceding tableif the first operand is **False**, the operator returns **False**, even if the second one is **True;** thus you don't need to evaluate the second operand and the procedure "short circuits" the comparison. The best way to understand this is through code.


```
Module LogicTest
  Sub Main()
    Dim x As Integer = 1
    Dim y As Integer = 1
    If A(x) Or B(y) Then
      Debug.WriteLine("x= " & CStr(x) & ", y = " & CStr(y))
    End If

    If A(x) OrElse B(y) Then
      Debug.WriteLine("x= " & CStr(x) & ", y = " & CStr(y))
    End If
  End Sub

  Function A(ByVal v1 As Integer) As Boolean
    v1 = v1 + 1
    Return True
  End Function

  Function B(ByVal v1 As Integer) As Boolean
    v1 = v1 + 1
    Return True
  End Function
End Module
```

Copy and paste this code into Visual Studio or build and run the **LogicTest** console application in the Vb7cr solution (see the Introduction for instruction for downloading this demo). Insert a break point in the code at the following line:

```
If A(x) Or B(y) Then
```

similar to the **GetMessages** demo application discussed earlier. The menu lets you choose to return a decimal value in its binary form using the shift operators to populate a bit−mask. You can also choose to shift left or shift right a decimal value and simple return the decimal result.

You could have arithmetic exceptions in these operations so we have enclosed the calling methods between **Try . . . Catch** blocks.

```
Imports Vb7cr.BitShifters
Module SeeBits

  Private inPut, byShift As String
  Private outPut As Integer
  Private isCompleted As Boolean = False

  Dim E As BitShifters

  Sub Main()
    Private menuChoice As String
\
    While Not isCompleted
      Console.WriteLine("                         ")
      Console.WriteLine("----------MENU-----------")
      Console.WriteLine("-------------------------")
      Console.WriteLine("a: Decimal to Binary.")
      Console.WriteLine("b: Left Shift.")
      Console.WriteLine("c: Right Shift.")
      Console.WriteLine("d: Anything else to end.")
      Console.WriteLine("-------------------------")

      Console.Write("Choose a process: ")
      menuChoice = Console.ReadLine()
      Select Case menuChoice
        Case Is = "a"
          DecToBinDemo()
        Case Is = "b"
          LeftShiftDemo()
        Case Is = "c"
          RightShiftDemo()
        Case Else
          isCompleted = True
      End Select
    End While

  End Sub

  Public Sub DecToBinDemo()
    Console.Write("Enter a number to convert from Dec to Bin: ")
      inPut = Console.ReadLine()
      If Not (inPut = "") Then
        Console.WriteLine("")
        isCompleted = ProcessInput(inPut)
      Else
        isCompleted = True
      End If
  End Sub

  Public Sub LeftShiftDemo()
    Console.Write("Enter a number to shift left: ")
    inPut = Console.ReadLine()
    Console.Write("How many shifts left?: ")
```

Table 5−9: Resulting Decimal and Its Corresponding Binary Representation After Shifting Numbers by 1 or More

Left shift	Yields Decimal	and Binary
1 << 1	2	10
2 << 1	4	100
4 << 1	8	1000
8 << 1	16	10000
16 << 4	256	1 00000000

In the **getKibbles** method, the first operation is to assign 1 shifted left by 31. The shift operator moves the 1 bit to the left 31 times and then drops it at position 32 (to give 32 bits). The positions to the right of this 1 bit are then filled in with zeros and we end up with a mask (**kibbleMask**) of 32 bits as follows 10000000 00000000 00000000 00000000.

Next we use a very useful class that you will learn about in Chapter 15, the **StringBuilder** class. This class allows us to build a string "buffer," which appends to and grows the string as we need (the standard **String** object is immutable and thus useless for something like this).

The next important piece of code is the for loop which for 32 loops does a bitwise **AND** of the value passed into the method against the mask. If the left−most bit **AND**ed yields 1, then 1 is appended into **sBuild;** otherwise 0 is. After each comparison, the loop left shifts the value variable by 1 (**value** <<=1).

```
for (int s = 1; s <= 32; s++) {
  sBuild.Append( (value & kibbleMask) == 0 ? '0' : '1');
```

Then at the end of the loop all we need to return is the string value in the **sBuild** object by calling its **ToString** method.

```
return sBuild.ToString();
```


The utility of both the bitwise operators and C#'s shift operators can be used in a single application. For example, you can design the algorithm around decimal numbers and then use the C# shift operators to move the decimals right or left as needed. You can then use the bitwise **And**, **Or**, and **Xor** to control program flow on the literal value of the numbers. Thus, if you shift "1" to the left by 1 you get 2, and if you shift "2" to the left by 1 you get 4, as seen in Table 5−9 above.

Specialized Operators

The .NET Framework defines a number of specialized operators. I have provided some light coverage in this chapter to introduce them. More examples are available in the chapters listed in Table 5−10.

Table 5−10: Specialized Operators

Operator	Description	Action/Usage	See Also
Is	Compares objects	*Result = objectX* ***Is*** *objectY*	Ch. 8 and 9
Like	Compares string patterns	*Result = String* ***Like*** *Pattern*	Ch. 10
TypeOf...Is	Tests for the type of object	*If (TypeOf Object Is) Then*	Ch. 6, 8, 9, 10, 11, 15

```
E = M *(Pow(C, 2))
```

or just

```
E = M * C * C
```

Operator overloading is especially important in endowing a language with high−end mathematical or numerical computing abilityfloating−point operations. It is often the very high−end capabilities that elevate a language to critical acclaim. Java, for example, has struggled to penetrate the finance, statistics, and floating−point software markets precisely because it lacks operator overloading.

Not all .NET operators can or should be overloaded, however, because certain ambiguities may arise, which will only complicate rather than simplify development. For example, the compound assignment operators are split into two operators at the lower level, so there would be no way to change the definition of one or the other in the combination; besides, there would be no reason to. On the other hand, once operator overloading is made available to Visual Basic programmers, operators like Pascal's :=, might find their way into Visual Basic classes.

Visual Basic .NET currently does not provide the developer with the ability to overload any operators. We mention this for several reasons:

- C# does support it and that will make many Visual Basic programmers curious. Language interop is important in .NET and C# will often be used as an extension to Visual Basic and vice versa. Visual Basic programmers might switch to C# one day, specifically to develop a value type or some other class that overloads an operator (remember, even the floating−point "primitives" are objects).
- Controversy surrounding operator overloading has long preceded .NET and is worth investigating.
- Visual Basic programmers may be frustrated with Microsoft, as many Java programmers are with Sun Microsystems for not allowing operator overloading in Java (see note).
- Microsoft has added some very powerful features to Visual Basic that without operator overloading may seem somewhat emaciated (see Value Types in Chapter 8 and Multidimensional Arrays in Chapter 12).
- Operator overloading will at some point be possible with Visual Basic .NET.

  Note A number of independent software vendors have been fighting for several years to get operator overloading into Java, as part of an effort to improve Java's ability to handle advanced numeric and precision algorithms.

The debate over operator overloading centers on several issues. Its proponents claim that it is essential to doing complex numerical work in their language. Coding complex constructs can be very cumbersome without the ability to provide a custom operator. In many cases the standard notation and syntax elements are so complex and confusing that the best tack would be to use another language. A small percentage of Visual Basic programmers might think this way and then intermix C# classes in their code as I have in this chapter.

The opponents (in our case the Visual Basic .NET architects) believe (as Java's architects do) that operator overloading works against team projects and can make code harder to maintain. The following section makes the case for it facilitating complex numerics. (See Chapter 8.)

Table 5−12: Referenced Exceptions

Exception Object	Purpose

Chapter 6: Software Design, Conditional Structures, and Control Flow

Overview

The logical and typical flow of execution of a program is *top down*. Just as you read the words on this page starting at the top of the page and moving to the end, so too is it natural for execution to proceed in this sequential ordered manner. This is known as *sequential execution.* However, you can alter execution flowto provide choices that execute one expression or a block over anotherwith statements that either *transfer control* to other locations of the method or otherwise *branch* or *jump* from the line that it is currently on to another area in the routine.

During the 1960s and 1970s, overuse of this transfer of control created programs that were hard to maintain and debug. A user could typically jump from one place to another with many lines of code in between. The problems programmers faced were often blamed on the infamous ***goto*** statement, which was cited as a reason advancements in software engineering were slow from the mid−1900s toward the end of the 20th century.

If you are new to programming, **goto** lets you transfer control to any number of labels in a method. The more labels you have the more incomprehensible the program becomes. In the early days of computing, most programs were reams of unstructured code that were to software what a mile of ticker tape is to modern communications.

You may remember programming like that in Dbase: trying to find a bug required printing miles of code on tractor−fed printer paper (laser printers cost about $8000 then). The **goto** statements were like black holes in spaceonce you got sucked into one there was no way out.

The most important tenet of modern control−flow programming is to keep the line to be executed and the decision to execute that line as close together as possible. In other words, if you are going to execute a line based on a test that returns **True,** then that line should be the next line, not one that is 15 lines away.

Before the advent of structured programming, **goto** could transfer from line 15 to line 156443. Today, **goto** can only leapfrog short distances in the method, which is how it has been with classic Visual Basic. Now there is no reason for it to exist in object−based programming, or in the .NET languages. As a programmer, it's hard to see how **goto** is still with us for backward compatibility when so much legacy or classic code has to be rewritten.

Well−known engineers of the formative years of computer science, such as C.A.R. Hoare, Niklaus Wirth, Edsger Dijkstra, Corrado Bohm, and Guiseppe Jacopini, actively lobbied for the abolishment of **goto.** They held conferences and wrote many papers demonstrating the superfluousness of **goto.** Using flow diagrams similar to the ones in this chapter, they would demonstrate how alternative structured control/flow constructs, such as **Case** and **If**, could be used to control the sequence of execution and improve the readability and manageability of program code by an order of magnitude.

These constructs heralded the age of structured programming. Nevertheless, **goto** still made it into the BASIC language where it loiters to this day. Its inclusion in Visual Basic .NET is *not* auspicious to say the least, but we will discuss it again later in this chapter.

Before you embark on this chapter, keep this important rule in mind: The code in your methods should be organized as straight−line as possible. Avoid code that is anything but top−down. When you read this page

We will use flow diagrams to explain these elements.

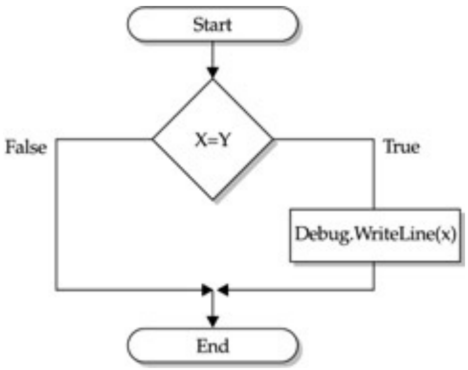


Figure 6−5: The **If** syntax flowchart

As we have seen, when a variable tests **True**, the statement after **Then** is executed. The syntax for the above flowchart is as follows:

If *condition* Then [Then *statements*]

And coding it is straightforward, as shown here:

```
If X = Y Then 'if condition is true then
  Debug.WriteLine("True") 'execute this statement
End If
```


When the condition to test is complex, use brackets to make the code more readable. However, be sure the expression is being properly evaluated according to the operator rules of precedence. See Chapter 5, "Operators." Here is an example of such bracket usage:

```
If (i > j * (j + z)) Then
  Swap(a(i), a(j))
End If
```

Stacking and Nesting If

You may often find yourself nesting or stacking **If** statements. The following example demonstrates the stacking of multiple **If** statements:

```
If (if a.Length > 0) Then
  pivotChar = leftSideIndex
End If
If leftSideIndex >= rightSideIndex Then
  swap(pivotChar, a(rightSideIndex))
End If
```

The above code is nothing more than independent **If** statements arranged one on top of the other. When you stack repeatedly like this, code becomes hard to read, is error prone, and might not perform as expected. When you need to test additional elements under the control of a single **If** construct, then use the **Else If** discussed later in this section.

Note Visual Basic .NET also supports a multiple−selection construct called **Select Case**, which we will investigate later in this chapter, so if you need to "stack" more than two **If** statements, your design may be better off with a **Select Case** statement**.** Also, there will be times when you need to construct a block that is executed will be the first one that matches the result of the complex call or computation. The following example demonstrates this:

```
Select Case Target.Position
Case TargetDown
'do something
Case TargetUp
'do something
Case TargetTurningIn
'do something
Case TargetTurningOut
'do something
Case TargetNoChange
'do something
```

You can use the new line symbol "**:**" to make the code easier to read for simple **Select Case** blocks as follows:

```
Select Case x
  Case 1 : x += 1
  Case 2 : x -= 1
End Select
```

The big difference between the **If. . . Else If** statements and the **Select Case** is that **If. . . Else If** can be used to test a number of alternative Boolean expressions in each block of the entire construct while **Select Case** examines only one and then uses the result to return the case paired with the **True** condition or run the code in that case.

GoTo

As mentioned earlier, **goto** (or **GoTo**) is still used in Visual Basic to help port code and assist with backward compatibility, though it is not essential to the .NET Framework. Since **GoTo** was one of the primary elements of classic BASIC and many other languages, programmers trained in these systems may find it difficult at first to write code without it.

It is surprising that C# supports **goto** because C# rarely needs to help port. This brings to mind only Java and J++ code, but Sun eliminated **goto** from Java so it wouldn't "pollute" the language.

Here's how it works. The **GoTo** keyword causes execution to jump from the current line to a label somewhere else in the block. The syntax is as follows:

GoTo *Label*

The following code illustrates the "disciplined" usage of **GoTo** branching. Anything more advanced would become too complex to understand and document.

```
Start:  str = Console.ReadLine()
        num = CInt(str)
        Goto Line0  'Check num and branch to its corresponding label.

Line0:  If num = 1 Then Goto Line1 Else Goto Line2
Line1:  Console.WriteLine("This is Line 1 and you typed 1")
        Goto Line3

Line2:  Console.WriteLine("This is Line 2 and you typed 2")
        Goto Line3
```

- Exit Try
- Exit Do
- Exit While
- Exit For

The same qualifying rule applies to exiting out of methods. You can exit directly from a **Function**, **Sub**, or **Property** procedure by specifying the type of method to qualify the Exit. The method qualifiers are as follows:

- Exit Sub
- Exit Function
- Exit Property

The method **Exit** statements can also be used inside the conditional and loop structures, for example inside **If . . . Then . . . Else** blocks. These are useful when you need to force a method to return at some point in your code.

Exit Idiosyncrasies

When **Exit** is encountered in nested control−flow or conditional structures, execution of code continues with the statement following the end of the innermost control statement of the kind specified in the **Exit** statement and execution returns to the previous level in the structure (**Exit** should not be confused with the **End** keyword). In the following example, **Exit For** is located in the inner **For** loop, so it passes control to the statement following that loop and continues with the outer **For** loop.


```
Public Sub InvertMyElements(ByRef myArray(,) As Double)
   Dim intI, intJ As Integer
   For intI = 0 To UBound(myArray, 0)
      For intJ = 0 To UBound(myArray, 1)
       If myArray(intI, intJ) = 0 Then
         Exit For
       Else
         myArray(intI, intJ) = 1 / myArray(intI, intJ)
       End If
      'this line is now processed when Exit For is executed.
      Next intJ
   Next intI
End Sub
```

You can also insert multiple **Exit** statements in conditional and control flow constructs. The following example shows this inside a **Do** loop:

```
Do Until level = desiredLevel
  If level <= desiredLevel Then Exit Do
    desiredLevel = CheckLevel(actualLevel)
Loop
```

The **Exit Do** statement works with all versions of **Do** loop syntax (with **While** or **Until**), and **Exit For** works with all versions of **For** loop syntax (with or without **Each**). Here is an example of **Exit** in all three places.

```
Sub CheckLevel(ByVal desiredLevel As Integer)
  Dim intI, level As Integer
  Do
    For intI = 1 To 5000
```

As the name suggests, constant data cannot be changed. It consists of read−only values, but the **ReadOnly** keyword is not valid in either of the method declaration spaces (see the "Properties" section later in this chapter).

Static Data

The **Static** keyword modifies a local variable declaration to static, which plays an important role in code reentrance, isolation, encapsulation, and recursion (see Chapters 4, 12, 13, and 14 and the later section "Recursive Design of Methods" in this chapter). When you declare static variables, their values are retained for future calls to the method.

It is critical to be aware that declaring a static local in a shared method means that the value of the static is retained for all references to the type. In other words, there is only one copy of the static local at all times. Dependence on the data held by the static must therefore be carefully reviewed. Remember that static methods (which are declared with the modifier **Shared** in Visual Basic and **static** in C#) are not instance methods. For all intents and purposes, the method and the static data are both global entities. (See the section "Improved Performance with Shared Classes and Modules" in Chapter 9.)

When you declare a static local in a nonshared method, which allows instantiation, then a separate copy of the static exists for each instance of the object, and the static's value is retained for the clients that have a reference on the object that encapsulates the static. The following code demonstrates declaring a static variable in a nonshared method (notice the use of Hungarian notation for clearly marking static variables):

```
Private Function ChurnOut(ByVal Param As Integer) As Integer
  Static stChurnval As Integer
  '...
End Function
```


Returning with Values

By default, all methods return to the caller or sender that called them. And as demonstrated in Chapter 6, you can use the **Return** keyword to terminate and exit out of a method at any point, even from a **Sub** method. In this regard **Return** works exactly like **Exit Sub**.

However, when you declare a function, you are advising the parties concerned that a value will come back from the method being called, so you must supply a return value and that value must be the same type as the value declared as the return value variable. This is demonstrated as follows:

```
Private Function ChurnIn(ByVal Param As Integer) As Integer
  '... do something with Param
  Return Param
End Function
```

The return value declared after the parameter list is a local variable declaration, just like the parameters and the variables declared in the body of the method. The function name is the name of the variable. For example, looking at the preceding method **ChurnIn**, you can see the variable declaration if you drop the parameter list as follows:

```
Function ChurnIn As Integer
```

To return **ChurnIn** as an **Integer**, you do not need to use **Return** unless there are several places in the function where return is possible (such as in a **Select Case** construct or a nested structure). However, if you do

You can avoid the problem by first avoiding hard−coding and doing away with magic numbers and arbitrary values in your code as shown in the following call:

```
GetVals(MyDayEnum.Sunday, MoneyToParamArray)
```

To send data to a parameter array your need to declare the parameter with the **ParamArray** modifier. And you cannot declare a parameter of type **ParamArray** without specifying the **ByVal (ByRef** is invalid)**.** Like the optional parameter discussed earlier the paramarray parameter must be the last parameter in the formal parameter list. Unlike the optional parameter you do not have to provide default values. If the sender does not send an argument to the parameter array the array will default to an empty array.

Parameter array usage is as follows:

- The parameter array will perform a widening conversion on the argument if the argument is narrower than the parameter. However, if the argument is wider than the parameter array or incompatible an exception will be thrown.
- The sender can specify zero or more arguments for the parameter array as a comma−delimited list, where each argument is an option for a type that is implicitly convertible to the element type of the paramarray. An interesting activity takes place on the call. The caller creates an instance of the paramarray type with a length corresponding to the number of arguments, initializes the elements of the array instance with the given argument values, and uses the newly created array instance as the actual argument to give to the parameter.
- Paramarray parameters may not be specified in delegate or event declarations.

Parameter arrays are useful and you don't need to treat the parameter array different, from inside your method, as you do the regular array reference.

Calling Methods

Methods are called (or invoked) via an interface, which is accessed by referencing the class or object containing the method, followed by a reference to the method's *signature*which is its name and (mostly) any arguments of the type and order, in the target method's parameter list. When you reference a method, you *invoke* it or *call* it. Some OOP experts also refer to the invocation of the method as "sending a message to the method." And thus the term *sender* is frequently used, especially in event models. Conversely, the method on the receiving end of the message or call is known as the *receiver*.

From time to time, we will refer to the construction of our code by the particular *method calls* that have to be made. Later in this chapter, we will see how Visual Basic methods can call themselves (recursion).

Call by Reference or Call by Value

As noted in Table 7−1, arguments can be passed to parameters by value or by reference. When you pass by value, you are passing the actual value to the method (some prefer to say "call by value"). When you pass by reference, you are passing only a reference to an object. Value typessuch as the built−in **Integer**, **Short**, **Single**, and **Double**are passed by value. Reference typessuch as arrays, strings, and custom objectsare typically passed by reference.

Suppose the method you are calling needs to receive an array. You don't need to send the array to the method (although this was once the case some time ago) but rather a reference to the array. This means the object stays exactly where it is and the method can still go to work on the array. This will become clearer to you in restricts access to the method to members of classes in the same application in which the method is declared, nested classes, and derived classes *and* applies the **Protected** access. The following code declares a **Protected Friend** method:

```
Protected Friend Sub StartInjector()
End Sub
```

Private Methods

The highest level of protection you can bestow on a method is achieved using the **Private** keyword. **Private**, as listed in Table 7−5, denotes that the method can only be accessed from within the class in which it is declared. However, **Private** methods can also be accessed from nested classes, because a nested class is part of the same declaration context or declaration scope of the **Private** method.

Composite or nested classes, which are discussed in Chapter 9, Chapter 13, and Chapter 14 are classes that are contained within classescompositionand thus they also have access to the private members of a containing class. The reverse, however, is not true. Members declared **Private** in nested classes are not accessible to members of the containing class, because the scope or declaration context does not include the container class itself.

Private methods, for example, can be accessor methods that compute data, or modification methods that set internal class data that may be required elsewhere in the class, possibly to be accessed by the consumer of the class as a static method call.

The following code declares a **Private** method:

```
Private Sub StartInjector()
End Sub
```


The **Private** modifier is similar to the **Protected** modifier in that a composite class can see a **Private** method if the method is shared. If the method is not shared the composite must collaborate with the outer class via an object reference in order to see the **Private** method.

Controlling Polymorphism Characteristics of Methods

The implementation characteristics of methods define their polymorphic characteristics, because they are declared as nonvirtual by default. There is a good reason for this: Nonvirtual methods cannot be overridden, so the compiler does not need to look ahead and figure out all the variations of calls that may be invoked, which methods they apply to, and so on. Static methods stay nonvirtual, which means the compiler can bind to the call at compile time, a process known as a *method inline.* Polymorphism (which means many forms) is discussed in more detail in Chapters 9, 10, 13, and 14.

However, polymorphism is a central tenet of object−based programming (see Chapter 10), and the .NET Framework allows methods to be declared as virtual, which means they can be overloaded and overridden. Overriding, for example, achieves polymorphism by defining a different implementation of a method (and a property) with the same invocation procedure. Table 7−6 lists the polymorphism modifiers, followed by the alphabetical explanation of each modifier (note that C# modifiers are lowercase).

The Methods of System.Math

Max	Provides the larger of two specified numbers.
Min	Provides the smaller of two specified numbers.
Pow	Provides a specified number raised to the specified power.
Round	Provides the number nearest the specified value.
Sign	Provides a value indicating the sign of a number.
Sin	Provides the sine of the specified angle.
Sinh	Provides the hyperbolic sine of the specified angle.
Sqrt	Provides the square root of a specified number.
Tan	Provides the tangent of the specified angle.
Tanh	Provides the hyperbolic tangent of the specified angle.

To investigate the constants and methods (and other members) of the **Math** class, open the Object Browser in Visual Studio. The easiest way to do this is to use the keyboard shortcut CTRL−ALT−J. The browser can also be accessed from the menus: Select View, Other Windows, Object Browser.

In the Object Browser, you need to drill down to the **System** namespace. As mentioned in Chapter 4, namespaces are preceded by the curly brace icon {}, while assemblies are represented by a small gray rectangle. Do not confuse the **System** namespace with the System assembly. **System**, the namespace, also lives in the mscorlib assembly, as illustrated in Figure 7−1.

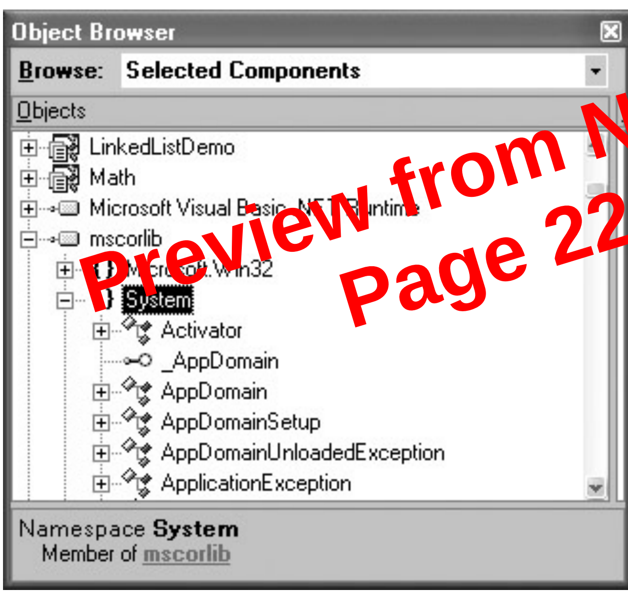


Figure 7−1: The System namespace in the mscorlib assembly

Expand **System** and you can scroll down until you find **Math**. Expand the class and the complete list of members will be loaded in the right pane in the Object Browser. Every method is documented, as illustrated in Figure 7−2.

The equation, using Pi, is $C = \pi * D$, where C is the unknown circumference and D is the diameter. Now we know that π is a constant of 3.14, so the circumference is 3.14 multiplied by 1,158. The circumference is therefore 3,636.12 kilometers (rounded to two decimal places). Let's write some code to express this:

```
Public Function Circumference(ByVal Diameter As Double) As Double
  Circumference = PI * Diameter
End Function
```

Simple enough, and we can glean more information about Ariel by also calculating its surface area. (These moons appear to have big chunks of ice, so if we ever run out of water on earth, we may need to put these planetary land surveying applications to work.)

The formula to calculate the area of a sphere such as Ariel is $A=4\pi r^2$ where A is the area of the planet.

This can be expressed with the following code:

```
Public Function Area(ByVal Diameter As Double) As Double
  Dim rad As Double = Diameter / 2
  Area = 4 * PI * Pow(rad, 2)
End Function
```

At approximately 1,053,191 kilometers, Ariel would be suitable for the next indoor Winter Olympics.

Here is the full listing of the **Math** demo:


```
Imports System.Math
Module Math
    Dim inPut As String
    Dim diameter As Double
    Dim Completed As Boolean

    Sub Main()

        While Not Completed
            Console.WriteLine("                     ")
            Console.WriteLine("------------MENU-------------")
            Console.WriteLine("-----------------------------")
            Console.WriteLine("Please enter the diameter.")
            Console.WriteLine("or press return to end.")
            Console.WriteLine("-----------------------------")

            inPut = Console.ReadLine()

            If Not (inPut = "") Then
                Console.WriteLine("                     ")
                Completed = ProcessMath(Convert.ToDouble(inPut))
            Else
                Completed = True
            End If
        End While
    End Sub

    'e=mc2 example
    Public Function E(ByVal M As Double) As Double
        Dim C As Double = 2.99792458 * (Pow(10, 8))
        E = M * (Pow(C, 2)) 'joules
        'same thing as E = M * C * C
    End Function
```

problem keeps getting smaller until it no longer exists.

In more complex problems, the algorithm knows how to solve the problem, but because the problem is so big, the algorithm divides the problem into smaller problems and then calls itself to go to work on the smaller problems. (Refer to the discussion of "divide and conquer" in the previous section.) This is why you often see array sort methods using recursion, as you will in Chapter 12, because the method partitions the array into smaller arrays and then sorts each one recursively.

The Stopping Condition

Every recursive method must have a *stopping condition* or the recursion will continue until the computer runs out of memory. In the preceding example, the stopping condition is when first and last become equal or land on the same index. At that point, the method must return (using the **IfThen** construct) or the two values will intersect, reverse the procedure, and run off the bounds of the array, causing the method to explode.

In this example, the stopping condition is placed at the point where we decide we have achieved the desired result. Running out of memory because the recursion continues on indefinitely is a worst−case scenario you must protect againstjust as you would with a **While** loop.

The method signature can thus be constructed as follows:

```
SwingArray(ByRef swinger() As Integer, ByVal first As Integer, _
  ByVal last  As Integer)
```

We pass the array reference, **first** (which is 0 or **swinger.GetLowerBound(0)**) and **last** (which is **swinger.Length−1** or **swinger.GetUpperBound(0)**). Inside the method implementation, we can swap the values as follows:


```
Module Module1
  Dim swinger() As Integer = {2139, 2432, 4391, 3432, 8932}
  Dim placeHolder As New Stack()
  Sub Main()
    SwingArray(swinger, swinger.GetLowerBound(0), swinger.GetUpperBound(0))
    PrintArray(swinger)
    Console.ReadLine()
  End Sub

  Public Sub SwingArray(ByRef swinger() As Integer, _
    ByVal first As Integer, ByVal last As Integer)
    If (first < last) Then
      placeHolder.Push(swinger(first))
      swinger(first) = swinger(last)
      swinger(last) = placeHolder.Pop
      SwingArray(swinger, first + 1, last − 1) '<−recursive call
    End If
  End Sub

  Public Sub PrintArray(ByVal swing() As Integer)
   Dim intI As Integer
     For intI = 0 To UBound(swing)
       Console.WriteLine(swing(intI))
     Next intI
    End Sub

End Module
```

The array is now reversed. Calling **PrintArray** provides the following output:

```
8932
3432
4391 <-f/l
2432
2189
```

Notice that we are using an **If** conditional because we don't need to loop inside the method. The recursive calls to the methodas markedtake care of the repeated runs through the code.

Of course, such recursion is really unnecessary, because a **While** loop (iteration) would handle the repeats. Here's the alternative using iteration:

```
Public Sub IterArray(ByRef swinger() As Integer, _
  ByVal first As Integer, ByVal last As Integer)
  While (first < last)
    placeHolder.Push(swinger(first))
    swinger(first) = swinger(last)
    swinger(last) = placeHolder.Pop
    first += 1
    last -= 1
  End While
End Sub
```

So, it should come as no surprise to you that you can write the recursive call with a **For** or a **While** loop. So why would you consider writing code that makes recursive calls? The first answer to this question usually comes in the form of a statement of surprise from many green programmers: "I did not even know there was any other way to do thisand I have loops that have completely lost their way."

But the first rule to consider is that if a problem can be solved effectively and quickly using loop constructs, then that should be your first choice. For most algorithms, loops are easier and quicker to write and are a natural component of your programming arsenal. Before you start thinking about moving a loop to a recursive call, explore tightening the loop by making it more efficient, choosing the correct operators, and so on.

Recursive method calls or algorithms, however, often offer us a natural and elegant way of dealing with a complex problem, and this is one of the reasons I brought the subject up in the first place. In Chapter 12 we are going to look at some data structures that can be elegantly manipulated with recursive algorithms; in some cases, recursion is the only way to deal with the problem.

The Impact of Recursion

You will find many algorithms that are inherently recursive and that may be better coded with recursion than loops. Keeping both the method and the size of the data structure being worked on small is very important, because recursive calls tend to impact the call stack, especially when the dataset explodes.

One of the worst reasons to use recursion would be to compute factorials or Fibonacci numbers. A good example (which I would never like to see in production code and thus will attempt to demonstrate) of such a case is processing the Fibonacci series:

```
"Start with 0 and 1 and then add the latest two numbers to get the next one
n: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 ...
Fibonacci (n): 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 ...
```

To understand the architecture, let's look at the roots of the issue. Objects are reference types that live in heap memory. When you create an object with the **New** keyword, it is placed in a heap−based memory location and you are given a reference to work with, rather than the object itself. This reference is a variable that holds a memory address where the bits of the object can be looked up.

Hence, the statement **obvar1 = obvar2** denotes that you are making **obvar1** and **obvar2** point to the same object. It does not means **obvar1** and **obvar2** are equal. We'll discuss the object−reference model later in this chapter.

Many software gurus, including Java's architects, believe object management is too cumbersome for essential software types, such as the elemental−data types. When you declare an **int** in Java, you are creating a variable that holds the data assigned to it and is stored on the stack, where it is processed more efficiently than your average object is. Java's primitives are not objects, and you cannot "talk" to them in reference type semantics.

Java's native types are powerful, slender, and fitalways there when you need them. Reference types are fat and lazy living on the heap, waiting for the garbage truck to collect them. But the .NET reference type object model is far from inefficient. Although heap memory is not as fast as the stack, which has a direct connection to the CPU, many object−oriented purists believe that Java's inventors erred greatly in NOT making the native types objects.

For starters, mixing primitive types with objects quashes polymorphism, because you can't place a primitive type in a field asking for an object. You first have to convert it into an object.

Also, primitive types cannot be easily deployed in an object model that provides runtime type information (RTTI) or reflection ability (see Chapter 2). In order to work with pure objects, we would have to first wrap the values manually in cumbersome wrapper classes, creating problems and setbacks in performance gains.

Currently, Java programmers must explicitly wrap an **int** every time it is needed in an object realm. A powerful contingent within Java is lobbying its creators to implement lightweight classes and convert primitives into objectsthus rendering Java 100 percent object−oriented.

The .NET architects benefited from this debate and adopted the lightweight class architecturealthough they are not divulging exactly why the CLR works so well. Was it possible to have it any other way? After all, the Common Language Specification (CLS) makes .NET the framework for all languagesexcept of course for C++, which, with its primitive type model and hybrid semantics, is far from being a pure object−oriented language.

This value−type model seems to provide the best of both worlds. It allows us to work with true objects on the stack and copy them to the heap when needed. We have the freedom to create new value types, which is a major benefit compared to the Java model, which even struggles with enumerations. The downside is the overhead of boxing, and only time will tell if the .NET inventors upped the ante on Java.

Structs and Enums Ahoy: Creating New Value Types

Let's now create our own value types, which are categorized in two groups deriving from the **ValueType** class specified by the Common Type System. They are called structures (or structs) and enumerations (which actually issue from **System.Enum**). The illustration shows the **ValueType** hierarchy and its two derivatives, which we'll discuss in this chapter.

Note The code for the above **Complex** structure is the ComplexTypes project in the Vb7cr solution.
Here's another illustration of a financial structure encapsulating a financial function found in numerous function libraries, like those in Microsoft Excel®. The following structure implements methods for computing financial information. I have just shown an attempt at the straight−line Double−Declining Balance formula (*book−value * 2/useful life*), which computes depreciation on an asset for a number of years.

The **DDB** function is computed iteratively. In the following code the book value starts out at a value minus the current salvage value (what the item can be sold for on Ebay today). The methods respectively return the amount the book value decreased in the specified period in its useful life and the amount of depreciation to report.

```
Imports System
Public Structure Accounting
  Dim cost, salvage As Double
  Dim life, period As Integer
  Dim factor As Decimal

  Public Sub New(ByVal rcost As Double, ByVal rsalvage As Double, _
    ByVal rlife As Integer, ByVal rperiod As Integer, _
    Optional ByVal rfactor As Decimal = 2)
    cost = rcost
    salvage = rsalvage
    life = rlife
    period = rperiod
    factor = rfactor
  End Sub

  ReadOnly Property DDBValue()
    Get
      Dim book As Double = cost  salvage
      Dim deprec As Double
      Dim year As Integer = period
      While year > 1
        deprec = book * factor / life
        book = book  deprec
        year −= 1
      End While
      Return book
    End Get
  End Property

  ReadOnly Property DDBDepreciation()
    Get
      Dim book As Double = cost  salvage
      Dim deprec As Double
      Dim year As Integer = period
      While year >= 1
        deprec = book * factor / life
        book = book  deprec
        year −= 1
      End While
      Return deprec
    End Get
  End Property
End Structure
```


The **Accounting** value type can be used as demonstrated here. The following code:

You can also reference other structuresany one of the fundamental value types, for startersin yours. Let's examine how the method **ClRed** in the following code:

```
Private colorDefault As Color
Public Sub ClRed()
  colorDefault = System.Drawing.Color.Red
End Sub
```

sets the default color to **System.Drawing.Color.Red**, which is itself a structure provided by the **System.Drawing** namespaces.

Passing Structures to and from Methods

You can pass structures as arguments to method parameters **ByValue** and **ByReference**, just as you can with any value type. This is critical, especially for returning arguments from methods. While you can pass several arguments to the formal parameter list of a method, you cannot return more than one value type or built−in data type.

There are many situations in which you can return a value type that sends more than a single value *as a single value*notwithstanding the oft−cited rule and condition that you can and should only return a single value from a method.

Structures Can Reference Objects

Structures can reference objects, even collection objects like arrays. The upcoming example includes objects and other structures:

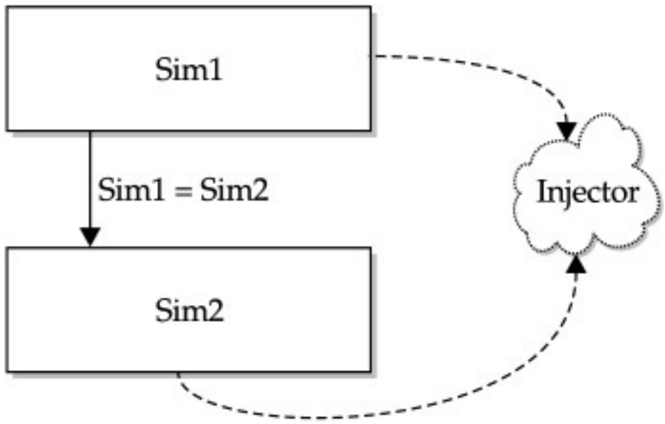
```
Public Structure Target

  Private targetColor As TGridColors 'target colors
  Private targetPosition As TGridCoordinates 'x,y positions on the grid
  Private targetSpeed As TSpol 'significant percentage of lightspeed
  Private targetType As TCraft 'the type of craft
  Private targetDistance As TDistance  'distance from our ship
  Private targetVector As TVector 'is the target going or coming
  Private targetHistory As History

End Structure
```

Structure Constructors

You cannot initialize a structure's members in the declaration, but you can initialize its variables in the constructor. The following code, from the earlier **GridColor** example, initializes the **colorDefault** variable of **System.Drawing.Color** structure in the **New** constructor.

```
Private colorDefault As Color
Public Sub New(ByVal red As Integer, ByVal green As Integer, _
  ByVal blue As Integer)
  colorDefault = Color.FromArgb(red, green, blue)
End Sub
```

You are not required to provide a constructor for a struct, as you would be to create an object instance. Even if you provide the **New** method, you do not need to call **New** in the declaration. Please note: if you neither provide nor call a constructor, zero will be the default for the struct's fields. This also explains why you cannot



Let's see what happens when we introduce a third **Injector**:

```
...
Dim Sim3 As New Injector()
Sim2 = Sim3
If (Sim1 Is Sim2) Then
  Debug.WriteLine("Sim1 is Sim2")
End If
```

Is does not return **True** anymore, because **Sim1** and **Sim2** no longer reference the same object. However, **Sim2 Is Sim3** returns **True**. There is a quirk: as long as two or more variable references refer to the same object, they are considered equal. Also, **Null** references (**Sim1 = Nothing**) also return **True** when compared with the **Is** operator.

To compare the objects, you should implement the **CompareTo** method defined by the **IComparable** interface or bridge to a *comparator* (see Chapter 12). You will be able to write code here that compares the bits of objects rather than the reference variables. Chapter 10 provides an in−depth discussion of this subject.

What *Me* Refers To

When you have a class that can be instantiated multiple times, you'll find that the **Me** keywordan internal reference variableconveniently references the object from within its own instance space. From this viewpoint, everything is visible, yet still protected from the outside world. Here we model the **Injector** object calling its own **GetType** method:

```
Public Function WhatAmI() As String
  Return Me.GetType().ToString
End Function
```

Note **Me** is the same as the keyword **This** in C#. It is also not legal to use it in a class module.
As you will learn in the next chapter, there are limits to using **Me**. For instance, it is not valid in shared classes that cannot be instantiated.

Observations

Microsoft is not alone in implementing primitives as first−class lightweight objectsseveral other languages have taken the same approach, including ADA and Smalltalk.

I scrutinized Java's primitives earlier in this chapter and concluded that they are primitive, or native, types and not first−class objects like .NET's value types. You can place Java primitives on the heap using wrapper

Functional models are represented with data−flow diagrams that show the dependencies between values that are computed, as output values, from input values. A functional model does not necessarily represent how the values are computed. It is not concerned with the inner workings of classes and methods or how the methods are executed.

Model Relationships

While each model alone describes various aspects of a software system, the combination of all of them, with references to each other, fully describe the software system. For example, the operations in the object model relate to events in the dynamic model and the functionality in the functional model. The dynamic model, on the other hand, describes the control structures of the objects in the object model. And the functional model represents the functionality that is achieved by the operations in the object model and the actions in the dynamic model.

It is important to understand that the models you create can never be exact representations of the actual software system. There is an accepted deficiency level because no model or abstraction can capture everything about the actual system or thing being modeled. Remember that the goal is to simplify the construction process and not burden it with overly detailed models.

Unified Modeling Language

Strange as it may seem, if you stop John or Jane developer in the lunchroom and ask him or her what modeling language they use, chances are they will think you are nuts because modeling is still not something a programmer considers important. This is especially the case with Visual Basic programmers, because classic Visual Basic as a language has never really lent itself to requiring such discipline in engineering. This is beginning to change in a hurry, because Visual Basic programmers now have full membership to the object−oriented club and are expected to have the correct disciplines. This is one of the reasons I decided to introduce this chapter with a backgrounder on modeling and modeling languages.

The visual modeling techniques we just covered are supported by an underlying modeling languagesupported by standardsthat a number of modeling tools support. When modeling software systems, if you cannot convey the model to interested parties the model will not mean much or be very useful. A visual model of a software project is not like a wooden model of a boat that is easily interpreted by physical look and feel. So, the software−engineering world came up with several notations over the past few decades, the most popular being the Unified Modeling Language (UML).

Visual modeling tools like Visio and Rational Rose support the three aforementioned notational or modeling languages. UML, however, is now by far the standard that has become the most popular. It is supported by austere governing boards such as ANSI and by the Object Management Group (OMG).

Over the years, object−oriented analysis, design, and modeling have relied on the collaborative efforts of a gang of wizards from several technology havens, especially Rational Software Corporation. The wizards include Grady Booch (Chief Scientist at Rational), Dr. James Rumbaugh, Ivar Jacobson, Rebecca Wirfs−Brock, Peter Yourdon, and several others. In particular, Booch, Rumbaugh, and Jacobson, the so−called "three amigos" that work at Rational, can be considered the caretakers of UML, and continue to work on the refinement of the language.

UML comprises a system of symbols that you use to build your software models. The symbols are similar to the Booch and OMT notations. UML has borrowed the notation elements from other notation languages, as well.

contain static methods and static fields, which means that all the members are shared. Modules cannot be instantiated (so there is only one copy of each field) and all data is global.

Programming for modularity in OOP is, however, just as important as it is in the procedure−oriented world. Encapsulation, one the founding principals of OOP, depends on modularity. In OOP, however, encapsulation is concerned with both information− hidingthe maintenance of secretsas well as the hiding of methods behind public interfaces.

Modularity Metrics: Coupling and Cohesion

So, if programming for modularity is so desirable, even for OO software design, how do we know that our classes are inherently modular? It's simple really. We just have to follow the two most important metrics of modularity*coupling* and *cohesion.* The coupling and cohesion metrics were discussed in some depth in Chapter 1, and if you missed the boat back then, you may want to return for a refresher.

Coupling

It is worthwhile repeating here that strong coupling detracts from the benefits of OO design because it makes the overall system harder to understand and maintain. When classes depend on each other for data and functionality, they become tightly coupled and this should be avoided. This is especially important when designing a system of objects, because tightly coupled objects detract from concurrency, reentrance, persistence of objects, and other such desirable traits (and benefits) of object−oriented systems. It becomes harder to maintain and understand classes the more dependent they become on other classes.

You should know that the coupling metric has a vital contraindication in OOP inheritance. The concept of inheritance denotes a hierarchy of classes where children depend on parents for their inheritances, data, and implementation.

Inheritance classes are thus tightly coupled, however, the loose coupling metric is elevated to the class hierarchy or the family. We will talk more about this in the "Inheritance" section later in this chapter.

Cohesion

The cohesion metric also came to life in structured design and is a critical principal of procedure−oriented software development. While coupling covers the relationship between classes, cohesion covers the degree of connectivity between the members of classes and their data.

Cohesion, discussed in Chapter 7, applies equally to all members of classes as well as the collection of methods within them. Strong cohesion among the elements of classes is what we strive to achieve.

The best−constructed classes are the ones that avoid coincidental cohesion, in which you just toss unrelated elements into a class. As discussed in Chapter 7, our aim is to construct classes that are strongly cohesive (functional cohesion), in which methods and data are exactly all the class needs to fulfill its role and duties and no more.

The Classes Are the System

When you think about your application as a system and not as a huge collection of "function points," it become possible to see the bigger picture and not be mired down in the minute details that can be so debilitating. For example, I have been working on a spacecraft simulator and can vouch for how quickly you logical unit, not as a collection of tightly coupled classes, the one depending on the other like two conjoined individuals. As long as you stick to the information−hiding/encapsulation recommendations and practices described later in this chapter, you will never see a detrimental result created from the inheritance mechanism.

Inheritance can actually detract from the encapsulation you have taken care to implement in your class. As an example, imagine that you decide to extend a class and use a method or some data as is from the base class. Now the class providera neat freak who just keeps improving his or her classesgoes and makes a change and reissues the assembly you are referencing (of course, that neat freak could be you), and now you have a problem. Because of the direct inheritance, the change ripples down the class hierarchy like a long line of dominoes. At the end of the line is your application, which gets knocked over.

Sounds like a big problem, but it's not really if you know what you are doing. In properly and carefully designed applications, you use the ability to override base functionality wisely. If you extend a class and absolutely need to depend on a new implementation in the child class, overriding effectively stops the domino ripple in its tracks. We will see how this works later in this chapter.

You can't override inherited variables and constants derived from on high. But any class designer worth more than a pound of salt is not simply going to change an **Intege**r you are using to a **Double** or a **Decimal**. Chapters 2 and 4 illustrated just how type safe the .NET Framework can be. With the correct configuration, it is very difficult to make changes without Visual Studio stopping you dead in your tracks. Despite that, you should shadow data fields that have the same name in parent classes, or declare new variables and constants in the child classes.

The coupling effect of inheritance no doubt has to be considered. It is also possible to change implementation or add override functionality along a deep hierarchy, which can result in some nasty conflicts. A cohesive development team implementing a framework will be able to manage the process with common sense. In other words, you still have to be careful.

If you don't intend your derived classes to be further derived or you are getting ready to implement your derived classes for the greater good of the application, then methods and other implementation can be sealed or made final, thereby preventing other users of the class from further overriding your methods. We will delve into this in more detail later in this chapter, after we have reviewed all the various ways of constructing classes, the roles of classes, and the relationships among classes.

Multiple Inheritance

Mother Nature is much more intelligent than any guru writing software is. She can easily fashion new life from the genes of more than one parent. For example, Laila Ali might punch like her famous daddy, Mohammed Ali, but the world knows that she also has her mother's looks.

Multiple inheritance (MI) allows a design and implementation concept known as a *mixin* in OO parlance. A mixin would allow us to inherit from more than one class and thus inherit the definition and implementation from the mixin. This is illustrated in Figure 9−9, where the new subtype of two or more parents contains the inherited elements of all the mixed−in classes.

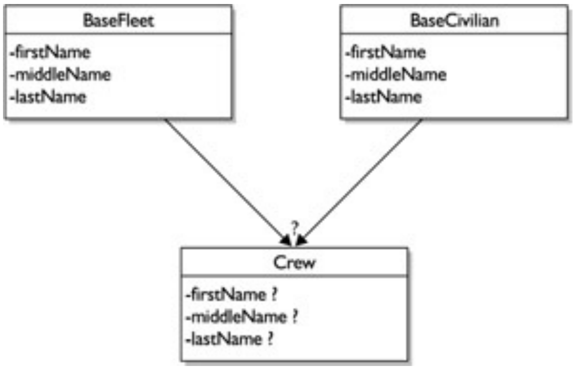


Figure 9−9: Multiple inheritance

MI in software, some believe, is too problematic for us rank−and−file software geeks, so we can't do mixins. The .NET type system thus only supports inheritance from a single parent. But it turns out there is good reason. MI adds to the complexity factor, which goes against what we are trying to achieve with inheritance in the first place.

One of the most common problems encountered with MI deals with identical method signatures that derive from more than one class. The problem you have to face when you derive from two or more parents with identical methods is determining which method to implement?

The purest form of MI lets a subclass inherit fields and implementation from different parents at the same time, and many class providers feel that the added flexibility and power is worth the extra care required during implementation. C++ changed to MI long after the language was introduced. Eiffel was built from the ground up using MI. Languages like Java and Delphi have opted for single inheritance only. This is the case with the .NET languages. (If you try to add a second **Inherits** statement to your class, the compiler will politely tell you to get lost.)

But single inheritance does not necessarily mean you only have one super or parent class. It means that inheritance can only be implemented through a single object hierarchy. While a language like C++ has multiple object hierarchies, the .NET languages only inherit from one hierarchy. The root **Object**'s members always manifest in every new class. So, a child class derives not only from your new custom base class, but also from **Object**. You can by all means derive from your custom class, and thus you would have a new child class that contains elements of three superclasses. This is acceptable (if not overdone) as long as there is only one logical hierarchy.

Order and Control with Inheritance

Classification provides order and control in software development projects, which so often becomes a chaotic situation. I have been involved in many extensive software development projects over the years, from classic applications such as highly efficient state machines/schedulers for telephony systems and telephone switches, to business applications such as accounting systems and CRM applications, to multimillion dollar e−commerce sites. In all of these projects, I have seen how quickly a team of developers can lose control over their code.

Classification of classes into hierarchies provides a means of order and control. It is a good idea to assign the responsibility of base class creation to a single developer or a group of developersclass providersand enforce the inheritance and extension of subclasses at the class consumer level, with the developers who need to use the classes.

Figure 9−10 shows how a chain of command is established for the class. Consumers know what they need to do to use the class in the application, and the providers know what they need to do to maintain the base classes. When consumers and the architects require new common class members to be added to the base classes, that responsibility falls back to the developers maintaining the classes.

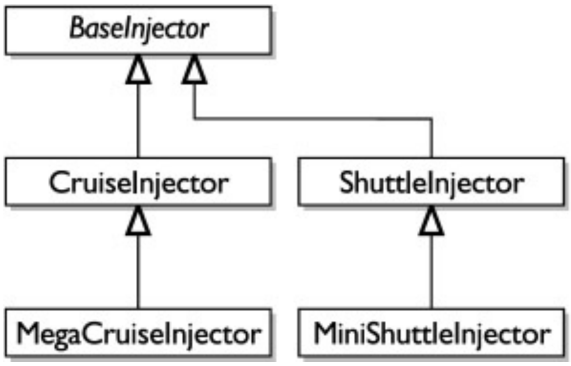


Figure 9−10: Control and ownership in class hierarchies

As the figure indicates, I propose providing class hierarchies for all classes in a project. The class hierarchies should thus evolve to become a framework, in the same context and for the same purpose as the .NET Framework. Some class hierarchies will be deep and lightly extended. Others might be shallow and heavily extended.

The creation of frameworks using inheritance thus implies a separation of objectives in the software development process. On the one hand, you are using inheritance to create new classes, polymorphism, and interfaces to feed the development, and provide new classes. On the other hand, developers use the framework to build their applications. I stumbled across an excellent analogy in a pottery shop at about the time I wrote this chapter. In this particular shop, you could buy the vase in its raw state, already shaped and baked, and simply paint it. So, the pottery shop provides the base and you provide the finishing touches without having to get your hands fouled up with sticky clay.

Reduction of Complexity

Developing class hierarchies and frameworks substantially reduces complexity. First, the class hierarchy is properly factored, as discussed earlier. So, documentation, adherence to models and specifications, interface usage and exposure, guidelines for deployment, and so on all become available to the team. Developing applications within a framework of classes that is well thought out and well documented is much easier than developing them within a hodgepodge of code that is just stuffed into classes like leftovers stuffed into Tupperware boxes.

At the same time, creating the framework provides a benefit that is obtained implicitly and without huge effort on the part of the project manager. No single person becomes the indispensable keeper of his or her "code" either through some desire to protect some interest or because a class is created on−the−fly and on−the−quiet. Often, classes are written as an afterthought, or for some other reason known only to the creator, without inclusion in the model or the specification, or without any forethought or inclusion or agreement from any other team members and the project manager.

It thus becomes much more difficult to lose control over documentation and source code when code is classified and admitted to a framework of class hierarchies. Well−written source code does not have to be fully documented to the extent that the developer writing the code explains in plain English (or any other lingo) exactly for what reason every variable, constant, method, or property exists. This is especially true with OO code. The code is to a very large extent self−documenting, and if you can read Visual Basic .NET source code, you can comprehend what is going on in the class. (Contrast this to the classic BASIC code, described bugs is enormous. That's not code reuse; that's code misuse.

You might also stumble across recommendations to avoid inheritance completely and use aggregation and containment techniques. This is not sound advice either. Nothing in life is a perfect fit; there are no constants. In some scenarios, inheritance points the way; in others, aggregation or another pattern points the way. The problem is not that one technique is bad and another is good. The problem arises when you use the techniques for the wrong reason.

It is true that inheritance has been overhyped in many quarters; and as a result, rather than learning when to correctly use inheritance, novice programmers start using it everywhere. The problem is that when they finally grow up, they can't shake the bitter experience of having to redesign applications and learn new tricks, so they slam inheritance and tell you not to use it. That's a difficult pill to swallow when you work with a framework, like .NET, in which inheritance underpins the entire infrastructure.

Implementing a Space Ship's Fuel Injector Software

We've come a long way with the theory in this chapter, but now for some code that shows inheritance in action. We are now ready to implement the **ShuttleInjector** class, which will comprise the following elements:

- **Instance Fields** These are the variable and constant data fieldsof course, they represent objectsof our class. For every instance of the class, there will be a separate and totally private copy of the object's fields that can only be accessed from within the instance they are part of.
- **Instance Constructors** In this class, we only need one essential constructor, the **New** method.
- **Properties** Properties are implemented to obtain status information related to the injector's on/off state and current velocity, and so on.
- **Methods** A number of methods will be implemented from the base class.

The **Inherits** keyword in a class specifies a class to be derived from. In other words, the class intends to inherit the interfaces, methods, and fields of the base or superclass.

The following code demonstrates the new class for an object (a simulator) that can control an injector, about to inherit from the class **BaseInjector.** Visual Studio will report to you that you need to implement a lot more than just **New**, specifically as directed by the base class through the facility of the **Inherits** keyword:

```
Public Class ShuttleInjector
  Inherits BaseInjector

  Public Sub New()
    MyBase.New()
  End Sub

'there are methods to implement

End Class
```

In the preceding declaration, the **Inherits** keyword specifies that the derived class, **ShuttleInjector**, inherits the properties, methods, and any initialization data from the parent class, **BaseInjector**. However, understand that the use of the **Inherits** keyword does not circumvent any non−inheritable or non−overridable members in the base class. These remain sealed if that is what you intended. The inheritance will become clearer in the next section.

variable is global to the class, the instance variables are declared private, which thus hides them from being accessed by the outside world. This is the essence of encapsulationthe hiding of information (discussed in Chapter 1). If you really think about it, nothing else, other than the members of the class, needs access to these variables, and no matter what or where your code resides in the class, you always have access to the class variables. The visibility and scope of variables is discussed at length in Chapter 4, and in Chapter 13, which tackles the subject of encapsulation.

The code in your IDE window should thus now look like the following showing the methods and properties that are yet to be implemented:

```
Public Class ShuttleInjector
  Inherits BaseInjector

  Private warpSpeed As Integer
  Private warpDrive As Boolean
  Private injectorStatus As String = "Injector is Offline"
  Const C As Integer = 186355

  Public Overrides ReadOnly Property GetSpeed() As Integer
    Get
    End Get
  End Property

  Public Overrides ReadOnly Property MPS() As Integer
    Get
    End Get
  End Property

  Public Overrides ReadOnly Property DriveState() As Boolean
    Get
    End Get
  End Property

  Public Overrides Property InjectorState() As String
    Get
    End Get
    Set(ByVal Value)
    End Set
  End Property

  'Gentlemen start your warp engines
   Public Overrides Sub StartInjector()
   End Sub

  'Stop the warp engines
  Public Overrides Sub StopInjector()
  End Sub

  'Set warp speed
  Public Overrides Sub SetWarpSpeed(ByVal newWarpSpeed As Integer)
  End Sub
End Class
```


Visual Studio will stop complaining about implementing the abstract methods as soon as you have overriden all inherited definitions.

When the simulator is executed, the default warp speed will be reported as 0. Notice that this accessor does not have a parameter, and you will not need to test for any precondition. The data required by the accessor property is provided by the instance variables and is available to the members of the class. The **warpSpeed** data is private but not exactly hidden either. Notice that the property is public and thus can be called by any class (see the section "Class Characteristics" earlier in this chapter). Placing the variable behind another layer in the class, a property, can further hide the direct access to the already private **warpSpeed** field.

Remember, warp speed is (in our case) light speed plus a significant percentage of light speed. If warp factor, represented by the constant value *C*, is equal to one light year, you can easily calculate the miles per second (MPS) of, say, warp 5 (**WarpFactorEnum .ImpulsePlusFive**). So, you could write a property called **MPS** that computes and returns the value as MPS. Let's now do the specification and code for the method to convert the warp speed to MPS:

- **Property definition: MPS.** This read−only property gets the current speed in MPS.

  ```
  Public Property MPS()
  ```
- **Returns:** The property gets the current warp speed from the **warpSpeed** field that is global to the instance and converts the speed to MPS.

The code should be as follows:

```
'Get the MPS speed of the warp factor
Public Overrides ReadOnly Property MPS() As Integer
  Get
   MPS = warpSpeed * C
  End Get
End Property
```


Now we need to implement **DriveState**, which returns a value **True** or **False** for the current state of the warp drive (on or off) as set in the **warpDrive** field. The specification is as follows:

- **Property definition: DriveState** This property gets the current state of the warp drive. If the drive is on, then the return value as a **Boolean** type will be **True;** if the drive is off, then the return value will be **False**.

  ```
  Public Overrides ReadOnly Property DriveState() As Boolean
  ```
- **Returns:** The property returns **True** if the drive is on and **False** if the drive is off.

The code should be as follows:

```
'Check the state of the warp drive
Public Overrides ReadOnly Property DriveState() As Boolean
  Get
    Return warpDrive
  End Get
End Property
```

The last property to implement is **InjectorState**, which returns the current **String** value held by the **injectorStatus** field. The property may also be used to supply new data to the **injectorStatus** field. The specification is as follows:

- **Property definition: InjectorState**. This property gets the current state of the injector held in the field's **String**.

derived classes or **Me**, the current instant of the class. You will deal with cloning, deep cloning, and the **ICloneable** interface in Chapter 10. (By the way, the Java architects also spelled their **Cloneable** interface with an *e* in the middlewhich is not exactly correct. Interesting coincidence, or is it that all software architects can't spell?)

GetHashCode

A discussion of genetic cloning will almost certainly lead to discussions about DNA. When talking about the cloning of objects, the discussion will usually include the hash code subject. A simple definition of a hash code is that it is an integer key, created on the contents of an object, which can be used as a means of searching and sorting objects.

The **GetHashCode** method is used to implement hash tables, which are used for doing fast lookups by key. A hash table makes use of the key to increase the efficiency of searching and sorting objects, and there are various tried patterns for its use.

Every object in .NET produces a hash code, and the **GetHashCode** method implements a very simple hashing function that produces a simple key. In fact, everything you do generates hash codes, because everything in .NET is an object. The elements in a list of URLs in a browser have associated hash codes, a collection of IP addresses have associated hash codes, and the elements in an array have associated hash codes. Every .NET programmer should have an unshakeable understanding of hash codes and hash tables.

In Chapter 12, you will look into what's involved in using hash tables, but for now, check out what the **GetHashCode** method retrieves. Adding the following line to your code,

```
Console.WriteLine(Sim2.GetHashCode())
```

writes integer "3" to the console (your compiler will probably return some similar number). How amazingly scientific is *that* result for one of the cornerstones of computer science?

You are probably thinking, "That doesn't look very unique either." It isn't. **GetHashCode** is another important method that is left up to the class implementors to override. The base class version simply returns an index value representing the class instance (the CLR chooses it, and not long after I tested it, it returned 3 for an **Object** that had already been disposed of), so it is very possible that it will not be unique. In fact, only the strongest hashing functions will produce a (relatively) unique hash code for you.

Making **GetHashCode** overridable is correct behavior, similar to the reason **Equals** is overridable. This mandates that you reimplement it using the hashing algorithm of your choice. This is the case with all OO languages. Stay tuned for more on this subject in Chapter 12.

GetType

You had a look at the **GetType** method earlier, in the discussion of the **Equals** method, so you know that it returns the exact run−time type of the argument. So, the statement

```
Console.WriteLine(Sim2.GetType)
```

just returns the FQON.

to exist as a separate class because tomatoes are used in many recipes.

The multiple inheritance (MI) would also muck up our code, even if it were possible in .NET. The first problem would be the clash of method signatures, because we would have two hierarchies deriving from **Circle** and **Cylinder**. So, think about the burger for a moment and not the code or the class. Tomatoes are placed inside burgers, between the patty and the bun. Why not just add the **Tomato** and the **Patty** class to a **Burger** class, just as you would in the kitchen?

This pattern (which is a technique) is called *aggregation.* As mentioned earlier, the application of the rule to determine when composition is valid is similar to the application of the inheritance rule. Instead of asking if a thing *is−a* thing, you ask if the thing *has−a* thing. In our case, the rule fits. A burger has a tomato or more, and has a patty or two; tomatoes do not have burgers, nor patties. And for that matter, our inheritance architecture is also sound because we know that a circle *is−not−a* burger. Remember, the rule states that aggregation (and composition) is represented by *has−a* relationships between classes. If **ClassX** *has−a* **ClassR**, then **ClassR** should be contained in **ClassX**.

You'll probably be surprised to learn that we have been using aggregation from the very beginning, even before we inherited one line of code. As discussed in the previous chapter and in Chapter 4, the fundamental types are also objects. So, we are really using the technique when we declare an **Integer** or a **String** object to reference the variable or constant data in our class. We are simply embedding these objects in our new class. Back to patty making.

We can now begin constructing our patty by referencing both **Patty** and **Tomato** classes in the **Burger** class. Once we have done this, adding the other classes, like **Pickle**, **Ketchup**, and **Onion**, should be a no−brainer. Adding methods for controlling calories and fat is a different matter, however.

The code for our **Burger** class can now be basically implemented as follows:

```
Public Class Burger
  Friend Patty1 As Patty()
  Friend Tomato1 As Tomato

  Private Sub GoBurger()
    Me.Patty1.Createpatty(2, 10)
    Me.Patty1.Location = New BunPosition(60, 140)
    Me.Tomato1.CreateTomato(1, 8)
    Me.Tomato1.Location = New BunPosition(60, 140)
  End Sub

End Class
```

If you now look at the code, you'll notice something truly incredible with OOP. Through both inheritance *and* aggregation, we are able to reuse all the previously implemented code for the benefit of the **Burger** implementation. We are reusing not only the code previously written only for the independent **Tomato** class, but also the code for any other "ingredients" of **Burger**. We will not implement the **Burger** class further because I am sure you now have the idea, and besides, replicator technology has not yet been invented (and when that day comes we will be ready).

It is also important to mention that classes that make wise use of composition patternsusing sealed classes as often as possiblewill provide much potential for the improvement in performance of your applications. See the section "Ending Inheritance with Sealed Classes," later in this chapter.

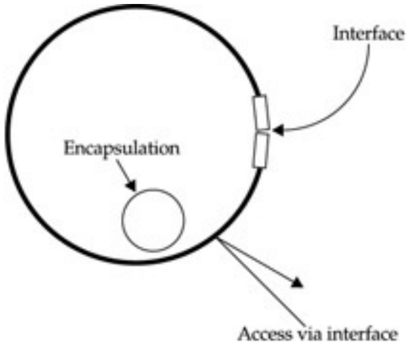


Figure 10−1: To access an object, you reference its interface

Getting Passionate (or Radical) about Interfaces

Interfaces are so elegantly integrated into the .NET Framework that I cannot help being passionate about the subject. Let's explore the reason for explicit interfaces further.

To fully understand object−oriented software development "philosophy" and its many concepts, you need to understand the concept of both abstraction and interfaces and why they are so important to .NET. Not only that, explicit interface design and implementation is a key requirement for "professional grade" .NET software developmentboth for standard applications and algorithms and for component development. Please don't think you can program without them.

Formal interface design and implementation is not new to software development, but it is one of its most misunderstood concepts. And, as mentioned in Chapter 2, interface implementation has often been blamed for DLL hell and versioning problems. The common language runtime's versioning features and side−by−side execution environment provide a haven for numerous versions of an interfaceeven if they are identicalthus freeing the programmer of the burden of maintaining versions of interfaces in a registry.

A large number of interfaces actually ship with the .NET Framework base class library. They carry no implementation, which is left to the programmer who can either implement directly or bridge the interface to an implementation that may already exist. Implementing an interface is a lot like implementing a Mini or some other car that everyone loves but that no one might be making any more.

Imagine walking into a store and buying just the Mini's chassis and the body and then being told by the salesperson that you now have to take home the shell and build your own internals, engine, seats, drive shaft, dashboard, and so on. You would probably be very confused at the prospectif you don't have a nervous breakdown at the idea. Why buy a car that *you* have to implement to drive?

Here's your dilemma: As a driver of past Minis, you've never really paid much attention to the implementation. It has always been abstracted away from you. You have had only to interact and communicate through the interface: the steering wheel, shift or gear stick, accelerator, and breaks. When something went wrong with the car, you took it to the shop, which knew the ins and outs of the "implementation." Now you are faced with all the messy details of the implementationthe idea of getting your hands all greasy is the last thing you were thinking about when you bought the "interface."

Looking at the interfaces in the base class library (and there are many of them), you, the programmer, might be excused for thinking the same thing. You design and implement a really cool application that the IT department is going to love you for, and a day before your are ready to deploy, the head of the financial applications department tells you she must have encryption support. No problem! The last time you checked in the huge collection of namespaces, there were encryption classes out the wazoo.

so on. In other wordsas demonstrated in the UML diagrams for inheritance in Chapter 9the communications officer on a spaceship *is−a* member of the crew. The parent class is thus **Crew**, and the subordinate, child, extended, or derived class is the **ComEngineer**.

The **Crew** class bestows all that is common about crewmembers to the descendent classes. **ComEngineer**, for example, will inherit the base members and either override, overload, or shadow the parent members.

Interfaces, on the other hand, do not allow inheritance of data and implementation, because (as we have discussed at some length) they have none.

It should thus be clear that inheritance promotes a tighter coupling of classes, while interfaces, by separating implementation from the interface, are able to promote loosely coupled and completely disconnected classes and the ability to access the implementation of unconnected objects. Most important, however, is that the interfaces drive poly− morphism in the system of classes and objects that makes up your applications.

Despite the confusion about interfaces being a substitute for multiple inheritance, it is clear where the misunderstanding originates. Suppose you need to provide a new object to represent a new type of crewmember on the spaceshipthe logistics crew. So, you create the class **Logistics** and derive from **Crew** to inherit all the common attributes and properties of **Crew**, such as **CrewName**, **CrewRank**, and **CrewID** fields, and methods used in authentication, sign−on, time on duty, and so on.

In the derived class, you extend the base class with members and implementation that applies to the **Logistics** class, even though the **Logistics** engineers are derivatives of **Crew**. Now you want to add support for comparing **Logistics** objects that compares fields unique to the **Logistics** objects.

The **Comparer** methods do not exist in **Crew**, so you'll have to implement the **IComparer** interface in **Logistics** instead or bridge to an implementation that already exists, to get the desired functionality available to the class. But is this a substitute for multiple or even single inheritance? No. How can the **Logistics** class, which we have said *is−a* crewmember, also be a child of the **Comparer** class? **Logistics** does not share an *is−a* relationship with the **Comparer** classin the same way that a *raptor* is not a member of the *canine* family, or a bicycle *is−a* train when it feels like going choo, choo, choo.

In other words, *inheritance* is used in object−oriented engineering only for two purposes (as repeatedly stated in Chapters 1 and 9): to represent the *is−a* relationship (and all of its benefits) among classes, and to express a tight coupling between the classes, for code reuse.

We also touched on multiple inheritance in Chapter 9. But multiple inheritance detracts from class structure. Even if you could inherit from multiple parents, the benefit of maintaining that focused class hierarchy would be very quickly lost.

Interface implementation and class inheritance, multiple or single, do share one thing in common, however: they both contribute to polymorphism between objects and methods. But interfaces *do not* make up for the lack of multiple inheritance in object−oriented languages, because they serve distinct and very different roles.

Thus, while the *architecture* for "inheriting" the definition of an interface is technically the same for standard class inheritance, proclaiming interface inheritance a substitute for multiple inheritance without understanding the difference is completely misguided and counterproductive.

```
    code = mdArray(CurrentCode)
  Catch Except As IndexOutOfRangeException
    Console.WriteLine(Except.Message)
  End Try
Dim EDA As New EventDataArgs(True, code)
OnEvent(EDA)
End Sub
```

The preceding method raises an **IndexOutOfRangeException** exception when it tries to access an element that is out of the upper bound of the **mdArray**. After the exception is handled, the execution continues after the **End Try** statement and the **OnEvent** call continues with 1 as the value for the code argument. In other words, **code** did not get changed.

What if you need to execute the code in the **Try** block that blew up? Handling the exception "gracefully" and then "leaving it at that" is not always enough. That's where the termination model used by .NET shines. You can always recall the method in the code after the **End Try**. Rerunning the code block can also be made possible from a **Finally** block that comes after the catch. Later we'll see how to use the **Finally** block.

It important to understand the two models, with the objective of writing code that is less buggy and easier to document and follow. Your error handling does not need to end in the **Catch** blocks of the method that caused you grief. The neat aspect of the underlying exception−handling architecture is that you can direct control through any **Catch** handlers until a handler that is specifically defined to handle the exact arrangement is foundno matter how deeply nested the source of the problem. As you'll see later, you can delegate the exception handling to another object entirely.

The exception handling process is analogous to a baseball game: After the pitch (the entry into the method), the errant ball is caught in the catcher's mitt and then thrown to second base to catch the runner trying to steal second. No luck at second base, so the defensive on second flings the ball to third base. Third is not tagged in time and the ball is thrown to home plate to catch the runner coming down from third.

While you do not need to "throw" an exception from one side of your application to another, you can use it to specifically rethrow or reraise the exception, even transfer it out of the original **TryCatchFinally** block to another method. You can also do whatever you need to do to fix the problem that caused the original hiccup and then return to the original method to try again (passing the ball back to the pitcher to have another shot). The exception−handling code and the code you can place in a **Finally** block can be used to roll back and clean up. It is key to remember that whatever happens in the exception−handling code, execution will resume with the code that comes after the **End Try** statement.

Recovering from Exceptions

There are exceptions from which you cannot easily recover. You can recover from your application or custom exceptions, but you cannot easily recover from most run−time exceptions without changing conditions in the system and hardware underpinning the application. What's the difference between exceptions raised from run time errors (in the CLR and even beyond its borders) and the custom exceptions?

Exceptions can be raised for the following reasons:

- **Syntax errors** These errors can occur if something is declared incorrectly and the compiler does not realize it. Syntax errors slip by unnoticed when syntax checking is turned off, by setting **Option Strict** to the **Off** position. A good example of a syntax error is an element or member of an object being accessed when the object has not been created.

- **Run−time errors** These errors occur during execution of your code, but may have absolutely nothing to do with your code. They can be produced by some of the simplest problems that may arise during run time, but the errors do not normally mean the algorithm or application is flawed. An example of such an error is an attempt to open a file or database that does not exist because the administrator moved the server. Your duty, however, is to write code that anticipates that a time may come when an idiot decides to delete a production database and bring the whole company, and your application, down. Other examples of actions that cause run−time errors include trying to dial a telephone number with no modem attached, serializing an object to a full disk, and processing a lengthy sort with no memory. In all of these cases, if the resources existed, no errors would result and no exceptions would be thrown. Run−time errors usually come from the operating system, which detects the violations in its part of the world, beyond the borders of the CLR where operating system services live.
- **Logic errors** These errors are similar to syntax errors because they go unnoticed by a preprocessor or the compiler. A *divide−by−zero* error is a classic example. This is not seen as an error until the program finds itself in a divide− by−zero situationthe logic of the algorithm leads the program to code that is essentially, but not inherently, flawed. Other examples include trying to access an element in an array that exceeds its upper boundary, reading beyond the end of a stream, trying to close a file that has not yet been opened, or trying to reference an object that has been terminated. Logic exceptions usually come from the operating system, which detects the violations. You may also provide custom exception classes to deal with your own logic errors.
- **Conditional errors** These are exceptions, usually custom−built by deriving from a base exception class, and are explicitly raised. You would raise exceptions only if a certain precondition or post−condition exists or does not exist in your code. For example, if a node of a custom linked−list class is not found at the start of an algorithm or block of code, you could raise a custom **NoSuchElementFoundException** exception to trap the condition. A post− condition exception would be raised if a condition you specify in the exception handler does not exist after the algorithm is processed. For example, if your code is supposed to leave the application in a certain condition before continuing and does not, you could provide an exception right therein a post−condition exception handler.

You can create custom exceptions to cater to anything you believe should be considered an error and that is not provided by the default exception classes provided in the base class library.

Note Using the directive **Option Explicit On** at the top of your class files forces the Visual Basic compiler to forward−check your syntax before it is compiled. It lets you be sure that all code is free of syntax errors before run time.

An exception is an object that is derived from the superclass **Exception**. When you add an exception handler to a method, you are essentially providing a means of returning control to the application and resuming execution as normally as possible. What would life be like if humans were provided with error or exception handlers like this? Just as you are about to make a gargantuan mistake, an error handler would catch the "error" and put you back on track. Humans learn from mistakes; unfortunately it takes a lot of effort to write heuristic software.

You can make it so that the caller of a method handles the exception raised in the target method. It might also be necessary for the caller of the caller to handle the exception, and you might have to go quite far back on the call stack to handle an exception.

When a method that bombs on an error is unable to deal with it, we say it has *thrown an exception.* This term comes from C++ and has caused many developers to balk at the idea of a class having a fit any time something goes wrong in a program. You might think of handling the "throw" as being similar to catching a ball at a baseball game. Drop the ball and miss the catch and you let the team down. Such exception handling is not a new idea. For example, structured exception handling has been part and parcel of the Object Pascal language and Delphi since its inception.

Exception Statements

To catch exceptions in Visual Basic, you need to enclose your code in a **TryCatch** block. The guarded code to execute is placed in the **Try** section of the block, and any errors are handled in one or more **Catch** blocks. After the last **Catch** block, you can provide the optional **Finally** block that will always execute regardless of whether or not an exception occurred. The **Finally** block is mandatory if no **Catch** block is used. Later, I will show you how you can use the **Finally** block to reset resources and provide some housekeeping.

Try

Back in the early '90s when I was a "newbie" Delphi programmer, I made an effort to code 99.9 percent of my stuff in **try except/try finally** blocks. In other words, no matter what routine I was writing, as soon as I arrived at the point in the method where the algorithm starts, the first line of my code was **try**. I am not ashamed to admit that I often did stuff like this:

```
       Try
  Y/0;
except
  on EZeroDivide do HandleZeroDivide;
end;
```


I did this (Object Pascal code) at a time when exception handling had just been introduced to the new object−oriented programming languages that were emerging. Clearly, more than a decade ago, many of us wrapped code in these exception−handling blocks just to "play it safe." You may laugh, but at least it was better than adding the line **On Error Resume Next** at the top of every routine regardless of what that routine did.

This habit carried over into my Java programming by 1995 and I always believed this until I decided to investigate exception handling in much more detail than I needed to. Before you get carried away, consider the following advice:

- Not all code produces exceptions. There's no point enclosing the call to **BackGround.SetColor** between a **TryCatch** block. First, you do not really have the ability to handle the exception properly, and second, it's unlikely that a property like this will be coded in such a way that it can risk exceptions. And even if a property were prone to exceptions, the exception handling should not be handled by a method that calls faulty code. Imagine asking a restaurant patron to "handle" his or her own "fly−in−the−soup" exception.
- Exceptions not handled by the method that raised them may, and often should, get handled by methods that came before it. In other words, the methods are popped off the call stack one by one until a suitable handler is found. So not handling the exception at the point it was raised does not mean your application is going to go to hell on the A−train. The **FindExcept** class presented later shows how this "delegation" works.
- Variables and constants declared inside the **TryCatch** blocks are only visible inside the block in which they were declared. In other words, their visibility does not extend beyond the scope of the guarded block of code. You will also not be able to see the variables from the **Catch** or **Finally** sections of the handler.

TargetSite

This property returns the method that raises your exception. You can combine it with some of the other properties, like **Source**, to supply your own information concerning the classes and method in which exceptions originate.

Finally

When an exception occurs, execution stops and control is given to the closest exception handler. This often means that lines of code you expect to always be called are not executed. There are times when resource cleanup, such as closing a file, must always be executed even if an exception is thrown. To accomplish this, you can use a **Finally** block. A **Finally** block is always executed, regardless of whether an exception is thrown and regardless of whether you used a **Catch** block to handle the exception.

The following code example uses a **TryCatch** block to catch the **IndexOutOfRangeException**s we have been looking at for the past couple of examples. In the following listing the **Finally** block executes regardless of the outcome of the action and sets the **isCompleted Boolean** variable to **False** in order to return to the menu rather than close down the application.

```
Public Sub Main()

  Dim menuChoice As Char
  While Not isCompleted
    Console.WriteLine("                     ")
    Console.WriteLine("----------MENU----------")
    Console.WriteLine("------------------------")
    Console.WriteLine("a: Test the NodeNotFoundException.")
    Console.WriteLine("b: Test exception parser")
    '...
    Console.WriteLine("i: Get a LookAtStackTrace.")
    Console.WriteLine("q: Q or nothing to quit application.")
    Console.WriteLine("------------------------")
    Console.Write("Choose a process:  ")

    menuChoice = Console.ReadLine()
    Select Case menuChoice
      Case Is = "a"c
        GetNodeNext()
      Case Is = "b"c
        TestParser()
      '...
      Case Is = "h"c
        LookAtSource()
      Case Is = "i"c
        LookAtStackTrace()
      Case Else
        isCompleted = True
    End Select
  End While
End Sub

'lots of methods in between

Public Sub GetNodeNext()
  Try
    Throw New NodeNotFoundException()
  Catch NExcept As NodeNotFoundException
```

Note Do not forget to make the derived class serializable with the **<Serializable()>** attribute (see Chapter 15 for more information on serialization).

The new exception class is tested with the following method:

```
Public Shared Sub TestNodeNext()
  Try
    myList.NodeNext
    If (NodeNext = Nothing) Then
      Throw New NodeNotFoundException()
    End If
  Catch Except As NodeNotFoundException
    Console.WriteLine(Exceptions.ParseExcept(Except))
  Finally
     'stay on current node
  End Try
End Sub
```

Observations

This chapter provided a thorough overview of exception handling in Visual Basic .NET because the subject is extremely important. It also served to supplement the introduction to exception handling in Chapter 7, which provided information on writing code with exception handlers as early as possible in this book. Exception handling is a vital facility in the design and construction of high−quality algorithms and robust methods.

An observation that I feel is imperative to point out in this chapter is that the pure inheritance in .NET comes into maximum use for creating your own user-defined, specialized, or custom exceptions. It makes perfect sense to derive from the base exceptions, as demonstrated earlier, because exceptions are all one of a kind. They all do the same thing and are tightly focused on handling exceptions raised in your code. Exceptions are thus tightly coupled and form a natural class hierarchy that is accessed by all parts of an application. It would make no sense, waste a lot of time, and cause a lot of anguish to your users and class consumers if you were to reinvent the wheel and develop your own hierarchy of exception classes.

```
Dim myStack As New Stack()
```

Tip Remember to use the **New** keyword or you'll end up with a **NullReferenceException**.
The following application demonstrates the important methods of the **Stack** class:

```
Imports System.Collections
Module Stacker

  Private inPut As String
  Private outPut As Integer
  Private isCompleted As Boolean

  Dim myStack As New System.Collections.Stack()

  Sub Main()
    Dim menuChoice As String
    While Not isCompleted
      Console.WriteLine("                         ")
      Console.WriteLine("----------MENU-----------")
      Console.WriteLine("-------------------------")
      Console.WriteLine("a: Push.")
      Console.WriteLine("b: Pop.")
      Console.WriteLine("c: Peek.")
      Console.WriteLine("d: Print.")
      Console.WriteLine("e: Find.")
      Console.WriteLine("f: Clear.")
      Console.WriteLine("g: Anything else to end.")
      Console.WriteLine("-------------------------")
      Console.Write("Choose a process: ")

      menuChoice = Console.ReadLine()
      Select Case menuChoice
        Case Is = "a"
          PushDemo()
        Case Is = "b"
          PopDemo()
        Case Is = "c"
          PeekDemo()
        Case Is = "d"
          PrintDemo()
        Case Is = "e"
          FindDemo()
        Case Is = "f"
          ClearDemo()
        Case Else
          isCompleted = True
      End Select
    End While
  End Sub

  Public Sub PushDemo()
  Console.WriteLine("Type something to push")
  inPut = Console.ReadLine()
    If Not (inPut = "") Then
      Console.WriteLine("")
      isCompleted = PushIt(inPut)
    Else
      isCompleted = True
    End If
  End Sub
```

```
        Return False
    End Try
End Function
```

Answer: EVOL is written to the console. This probably is not what you expected, but this is perfect for many operations that require you to store a chronologically acquired order of string objects. Here's an example that can be developed to keep track of the path a user takes through the Web site:

```
Public Sub MakeList()
  myStack.Push("http://www.sdamag.com/vb7cr/;$sessionid$QHDT1")
  myStack.Push("http://www.sdamag.com/ vb7cr /;$sessionid$AQBT5")
  myStack.Push("http://www.sdamag.com/ vb7cr /;$sessionid$AQBT6")
End Sub
```

The last item to go onto the stack is the last link the surfer came from. The preceding implementation is pretty straightforward. Often you get the most utility from a stack when you reference it from within nested, iterative, and recursive structures. The following code shows the pushing and popping from within the fabric of a recursive construct:

```
Public Sub Transpose (ByRef array() As Integer, _
  ByVal first As Integer, ByVal last As Integer)
  If (first < last) Then
    placeHolder.Push(array(first))
    array(first) = array(last)
    array(last) = placeHolder.Pop
    Transpose(array, first + 1, last − 1)
  End If
End Sub
```


Now, stand in line for a peek at queues.

Note **RemoveAt** is not implemented in the **Stack** class, which would defeat the LIFO utility of a stack. Remember, you can't pull a plate from the bottom or middle of a stack, or else you end up with a lot of broken china. However, you can pull a plate from the bottom or middle of a list. A **RemoveAt** implementation is covered in Chapter 13 in the Linked Lists and Trees section.

Queues

What's a queue? Or rather, what's in a queue? A queue is a FIFO software construct used practically everywhere to process items in an ordered fashion. A queue is the opposite of the stack, on which the elements ahead in the stack are pushed down. Figure 12−2 provides a simple graphical representation of a queue.

GetLength	Returns the length of a dimension
GetLowerBound	Returns the lower bound element of a dimension
GetUpperBound	Returns the upper bound element of a dimension
GetValue	Returns the value at a specified index in any dimension
IndexOf	Returns the index at the first occurrence of the value searched for
Initialize	Not yet implemented
LastIndexOf	Returns the index at the last occurrence of the value searched for
Reverse	Reverses the elements in a one−dimensional array
SetValue	Sets the value at the specified index in a one− or multidimensional array
Sort	Provides built−in sort operations
IsFixedSize	Returns **True** or **False** if the array size is fixed
IsReadOnly	Returns **True** or **False** if the array is read−only
IsSynchronized	Returns **True** or **False** if the array is synchronized (thread−safe)
Length	Returns the length of a one−dimensional array
Rank	Returns an ordinal representing the number of dimensions in the array
SyncRoot	Returns an object used to synchronize access to the array

Note Table 12−4 does not include the members inherited by **System.Array**, such as **GetType** and **ToString**.

Declaring and Initializing Arrays

When you create an array, you need to declare a length for each of its dimensions. The length of the array is the number of elements that you need to store. As you now know, .NET array elements (no matter the language) are referenced through a zero−based index, which, as mentioned earlier, means the first element of the array is given the index value 0. So, to specify the length of the array, you defer to the range of indices counting from and including 0.

Arrays are declared in the same manner as you declare any variable. Visual Basic array grammar includes parentheses or brackets after the data type (as opposed to square brackets used by other languages). In the following example, an array reference variable called **sAlarms** is declared, the intention of which is to reference an array of ordinal values:

```
Dim injectorAlarms(10) As Integer
```

If you need to declare an array reference variable that will reference **Double** value types, you could declare the array reference as follows:

```
Dim latinumPercentages(10) As Double
```

How many values can either of the preceding arrays hold? Tip: While they are declared as arrays of ten elements (0 to *n*−1), you'll be surprised to discover that you won't get ten. The Visual Basic architects did something here that confuses a lot of programmers. Using the **ForNext** iteration structure (discussed in the previous chapter), let's find out what gives:

```
Public Sub OffByTwoArrays()
  Dim injectorAlarms(10) As Integer
  Dim intI As Integer
    For intI = 0 to injectorAlarms.Length
    Next intI
```

```
  Dubug.WriteLine(intI)
  Debug.WriteLine(injectorAlarms.Length)
End Sub
```

Holy mackerel! The **injectorAlarms** array has 11 values (0 to *n*−2). Talk about getting what you didn't ask for. (What's worse is that **intI** after the **For** loop ends up at 12 (because it started at 0).) Why did the VB architects do this? To make it easier to convert from the 1−based arrays supported in the classic versions of Visual Basic (version 6 and earlier), under the covers, the Visual Basic implementation of the .NET arrays tacks on the zero element and thus adds one more element to the declaration of, in this case, 10.

When you convert a Visual Basic 6 or earlier array, you get the extra element over and above (or would that be "under and below"?) the original array lengthin the zeroth position. The problem is that C# and other .NET languages do not work that way. The same declaration in C# produces a ten−element array (0−9). To thus declare arrays to the exact length you specify in the declaration, you can use the following kluge code:

```
Dim latinumPercentages(10 − 1) As Double
```

It's not elegant but it will do until Visual Basic's array declaration works like C# or J# declarations, or we get a new array class that's not as confusing (or you take the bold step of creating a new array class from scratch).

The preceding lines of code thus declare the array reference variable named **injectorAlarms** with the "potential" to hold 11 **Integer** or **Double** values. In other words, the preceding code does not yet lead to the creation of the actual object. The object gets constructed implicitly at the attempt to assign variables to the declared elements and when you call **New** in the declaration. This means that we can create the array reference variable but delay initialization and activation of the array object, a tactic you can get away with when you set **Option Explicit** to **Off** to allow implicit typing declaration.

The constant between the parentheses does not have to be a number. Any legal means of obtaining the constant will do, even a value in a queue will work, as demonstrated here:

```
Dim latinumPercentages(ArrayQueue.Dequeue) As Double
```

The array reference variable should not be confused with the array element values, which are also variables (of types). The array variable is really nothing more than a reference to an array object, as you will learn about in the following chapter.

To initialize the array object in the declaration of the reference variable, you need to call the **New** constructor. The following example creates the array object **Alarms** to hold four **Integer** variables (in elements 0 through 3):

```
Dim Alarms() As Integer = New Integer(4) {}
```

What does this code do? As you are aware, an array in .NET is an object that derives from **System.Array**. The **New** operator thus accesses the constructor of the array class and passes the argument to create the array of five elements. The array can thus be illustrated as in Figure 12−5.

The Basics of Sorting Arrays

Most algorithms that use arrays will require the array to be searched for one reason or another. The problem with the code in the preceding section is that the array we were searching was at first not sortedand you saw the result. If the value we are looking for turns up at the end of the array, we will have iterated through the entire array before hitting the match, which means we take longer to get results because the binary search cannot perform the *n/2* operation. If the array is huge, searching it unsorted might give us more than unpredictable results.

Sequential searching like this will suffice when the size of the data set is small. In other words, the amount of work a sequential search does is directly proportional to the amount of data to be searched. If you double the list of items to search, you typically double the amount of time it takes to search the list. To speed up searching of larger data sets, it becomes more efficient to use a binary search algorithmor an *O(logn)* algorithm. But to do a binary search, we must first sort the array.

Search efficiency is greatly increased when the data set we need to search or exploit is sorted. If you have access to a set of data, it can be sorted independently of the application implementing the searching algorithm. If not, the data needs to be sorted at run time.

The reason array sorts are so common is that sorting a list of data into ascending or descending order not only is one of the most often performed tasks in everyday life, it is also one of the most frequently required operations on computers (and few other data structures can sort and search data as easy as an array).

The **Array** class provides a simple sorting method, **Sort**, that you can use to satisfactorily sort an array. The **Sort** method is static, so you can use it without having to instantiate an array. The following code demonstrates calling the **Sort** method statically and as an instance:


```
'with the instance method
With sAlarm
 .Sort(sAlarm)
End With
'or with the static method
Array.Sort(sAlarm)
```

The sorting method comes from the **Array** collection of methods. Simply write **Array.Sort** and pass the array reference variable to the **Sort** method as an argument. The **Sort** method is overloaded, so have a look at the enumeration of methods in the class to find what you need.

The following code sorts an array (emphasized) before returning the index of **Integer** value 87, as demonstrated earlier:

```
Public Function GetIndexOfValue(ByRef myArray() As Integer, ByVal _
  ArrayVal As Integer) As Integer
  Array.Sort(myArray)
    Return .IndexOf(myArray, ArrayVal)
  End With
End Function
```

While the **System.Array** class provides a number of **Sort** methods, the following sections demonstrate typical implementations for the various array−sorting algorithms, such as *Bubble Sort* and *Quicksort.* These have been around a lot longer than .NET and translate very easily to Visual Basic code. Porting these sorts to Visual Basic provides a terrific opportunity to show off what's possible with .NET, the **Array** methods, and built−in functions.

```
      End If
    Next inner
  Next outer
End Sub
```

What's cooking here? The **BubbleSort** now does two sorts in one method. It first sorts the first part of the array and then it sorts the second part. But before we look at the innards of the method do you notice that the stack of sorts seems a bit of kludge. The stack of sorts seems inelegant. It is. But trying to combine the two iterative routines into one iterative routine that still sorts the two parts separately is extremely cumbersome. If you look at the two separate sorts you will see how the method now lends itself to recursion. As we discussed on the section on recursion in Chapter 7, there are times when recursion is the best solution (and sometime the only one) when you need to use divide and conquer techniques to solve a problem.

However, designing recursion can be hairsplitting as well. You need to decide what gets passed to the method for the first sort and what gets passed for the second sort. Have a look at the following code. You'll notice we now have to pass variables to the method instead of fixed values. These variables cater to the start and end points at which to logically partition the array.

```
Public Overloads Sub BubbleSort(ByRef array() As Integer, _
  ByVal outerStart As Integer, _
  ByVal innerStart As Integer, _
  ByVal bound As Integer)
```

(**BubbleSort** must now be overloaded to cater to the multiple versions of this method we can come up with (we still preserve the original for simple sorts on small arrays).) The **outerStart** and **innerStart** parameters expect the starting position on the array for both **For** loops in the method. The **outerStart For** loops for each element in the array and the **innerStart For** loops for the number of comparisons that must be made for each element. The **bound** parameter expects the upper bound of the array part to sort to. The recursive method to sort the two parts can be implemented as follows:


```
Public Overloads Sub BubbleSort(ByRef array() As Integer, _
  ByVal outerStart As Integer, _
  ByVal innerStart As Integer, _
  ByVal bound As Integer)
  If outerStart >= bound Then
    Exit Function
  End If
  Dim outer, inner As Integer
  For outer = outerStart To bound
    For inner = innerStart To bound
      If (array(inner) > array(inner + 1)) Then
        Transpose(array, inner, inner + 1)
      End If
    Next inner
  Next outer
  BubbleSort(array, outer, inner, array.Length - 2)
End Sub
```

The recursive call is highlighted in bold. Before we continue, take note of the stopping condition (also in bold).

```
If outerStart >= bound Then
  Exit Function
End If
```

The pivot element is exactly thatpivotal. This sort is fast because once a value is known to be less than the pivot, it does not have to be compared to the values on the other side of the pivot.

This sort is faster than the sorts we coded earlier because we do not have to compare one value against all the other values in the array. We only have to compare them against *n/2* values in the entire arraydivide and conquer.

Exactly how fast is the quicksort? Taking its best case, we can say that the first pass partitions the array of *n* elements into two groups, *n/2* each. But it is possible to partition further into three groups, *n/3* and so on. The best case is where the pivot point chosen is a value that should be as close to the middle of the array as possible, but we'll get back to that after we have coded the algorithm.

Quicksort has been rewritten for nearly every language, and it has been implemented using a variety of techniques. The .NET Framework architects have also implemented it in C#, and it's a static method you can call from the **Array** class. But let's code the algorithm ourselves. Later, you can check out which implementation of quicksort works faster, the Visual Basic .NET one or the C# one.

To recap, the element that sits at the intersection of the partition is called the pivot element. Once the pivot is identified, all values in the array less than or equal to the pivot are sent to one side of it, and all values greater are sent to the other. The partitions are then sorted recursively, and when complete, the entire array is sorted.

The recursive sorting of the partitions is not the difficult part; it's finding the pivot element that is a little more complex. There are several things we don't know going into this algorithm:

- We don't know anything about the array elements and their values. When an array is passed to you for sorting, you don't get any advanced sorting information such as the best element to use as the pivot.
- We don't know where the pivot element may finally end up in the array. It could be in the middle, which is good, or it could end up close to or at either end, which actually slows down the sort (and, of course, that's bad).

So, to begin somewhere and without any information, we might as well just pick the first element in the array and hope that it's possible to move it to an index position that intersects the array as close to its final resting place as possible. But we still don't know where that might be (incidentally, you can also use the last element as the pivot). Let's look at another array to sort, represented here by its declaration and "on paper" in Figure 12−11 (it's easier now to represent the array horizontally):

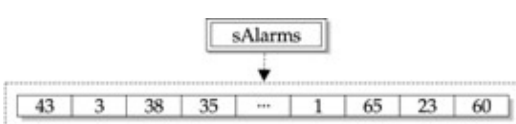


Figure 12−11: The unsorted array

```
Dim sAlarms() As Integer = {43,3,38,35,83,37,3,6,79,71,5, _
78,46,22,9,1,65,23,60}
```

This array is unsorted and yields the number 43 in the first element. So, we now need a method that will take all the numbers in the array less than or equal to 43 and move them to the beginning of the array. However, because we have chosen the first element as the pivot, we still don't know how far we need to move the less than or equal to elements to the one end of the array or how far we need to move the greater than elements to the other end.

The way this problem has been solved over the years is like this: Start at each end of the arrayfrom the element after the pivot (0) to the other end of the arraycomparing the value of each element with the value of

4	4	4	4	4
3	3	3	3	3
2	2	2	2	2
1	1	1	1	1
0	0	0	0	0

Figure 12−16: Accessing five arrays at the same subscript at the same time

The method you might create would be a real grizzly. Such a method can get very hairy and impractical when the length and number of arrays grow. This structure sometimes is called the "parallel array structure," but you don't need to name it, because such code is not what makes software fun, and you should forget this approach.

Could a better solution be to declare an array of six dimensions, as demonstrated in the following declaration? (Careful, this is a trick question.)

```
Dim InjectorStats As Integer = {New Date(), New DateTime(), _
New Integer(), New Integer(), New Integer(), New Integer()}
```

You are not going to get very far with the preceding **InjectorStats** array because of one fundamental limitation. Arrays are strongly typed, so the other dimensions must also be **Integer**s. Even if you could create a multidimensional−multitype array, this code is also exactly the definition of "inelegant." Really, it's not practical either.

This is where you need to put your object−thinking cap on. What if you did not store any of the actual values in the array and only stored references to objects? After all, that's how arrays of **String** work. By storing objects in the array, we only have to collect the reference variables. No injector data is stored in the array. Instead, we create an object that will contain the five data fields we require. Every reference variable stored in the array thus becomes a reference to the six data fields.

What do we achieve with this approach? First, we only need a simple one−dimensional array. No parallel arrays and no multidimensional arrays are needed. Second, the array can hold any object, as long as the objects are all of the same type. So we don't have a type mismatch issue with the array, which can continue to behave in its strongly typed way. Third, the data contained by the actual objects can now be processed and managed in any sophisticated way you may conceive. Fourth, your algorithm can be easily changed and adapted. The array of your custom type never needs to change, while the object itself can be extended and adapted as needs dictate.

The data can also be printed, sent to Crystal Reports and persisted through serialization (see Chapter 15), or stored in a database or a data mart. When we are finished with the objects, we can leave the garbage collector to clean up the bits left lying around. Not only is this all possible, but it's one of the most elegant ways to meet this type of data processing requirement.

First, we need an object that represents a record (like a tuple or row in a database table). We can call this object **Row**. While we can declare the container array wherever we need to work with **Row**s we need to create a class for the **Row** object so that **Row**s can be instantiated as needed (the number of **Row**s would be limited to the available memory). The class must have a constructor so that it can be instantiated with the following syntax:

```
Dim InjectorRows As New Row
```

Using object semantics, you can encapsulate six objects as fields of the container object. The first cut of our class might look like this:

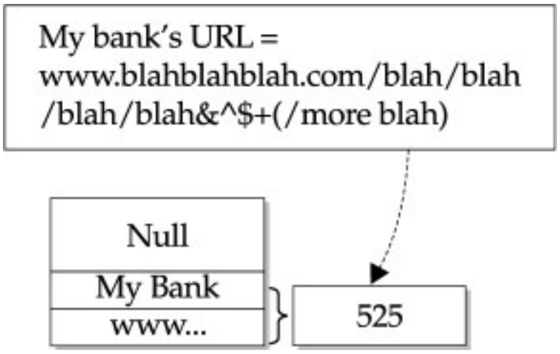


Figure 12−19: URLs represented by key and value in a hash table

But instead of searching on a string, no matter how simple, the name is hashed to an efficient **Integer** value, which becomes the key the hash table uses to look up the URL in the hash table. This is illustrated in Figure 12−19, which shows the associated **String** identifier of the URL hashed to its new key value.

Hash tables are considerably faster to search than standard binary searches, which require data in an array to be sorted (otherwise, the binary search will be meaningless). The underlying structure of a hash table has been implemented over the years in a number of different algorithms. Some implementations are better than others for certain types of information.

The hashingso−called because the value extracted from the process is a "hash−up" or "scrambling" of numbers that represent a stringproduces key values that are used to identify the location, or the subscript, of the hash table.

To avoid the collision of hash values that hash to the same location in the hash table, key values are organized in so−called *buckets* that associate the hash value with a list of items that share the key. A technique called *chaining* keeps the list of associated values together. The buckets and chains are shown in Figure 12−20.

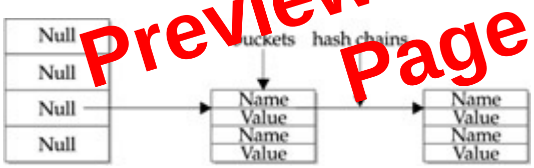


Figure 12−20: Hash buckets and chains

Why is a hash table, which typically operates at *O*(*1*), so much faster to search than a vanilla array? As we saw earlier in this chapter, in the section "The BinarySearch Method," we literally have to go through every element of the array to find what we are looking for. The only way to speed up the search is to partition arrays, sort the partitions, and scrounge for shortcuts, such as eliminating values that do not fall within the search range.

A hash table, however, is organized in such a way that when you pass it a key to retrieve, it knows exactly which bucket to look into. In the same way, we do this when we fetch the mail. We don't sift through a pile of envelopes from 50 companies; we just know to go look in the first mailbox.

In other words, the hash table only has to look through a subset of all the elements, which, for a binary search, is like not having to first sort and then partition the array, a process that would cut the time to find the element by an order of magnitude. By and large, you can think of the hash table as having knowledge of a shortcut to the data, whereas an array only knows the official route.

The **Hashtable** class is referenced in the **System.Collections.Hashtable** namespace and implements the following interfaces: **IDictionary**, **ICollection**, **IEnumerable**, **ISerializable**, **IDeserializationCallback**, **ICloneable, IList**, and **ICollection**. Thus, many of the methods (such as **GetEnumerator**) have been

Chapter 13: Advanced Design Concepts: Patterns, Roles, and Relationships

Overview

In Chapters 8, 9, and 10, we covered the core foundation or structural patterns of object−oriented software, such as interfaces, abstract classes, inheritance, aggregation and composition, and association. In this chapter, we will look at some advanced concepts in class design and implementation, as we keep these patterns, class relationships, and class roles in mind.

Our sojourn into these advanced class concepts will be allied with continued treatment of data structures and algorithms started in the previous chapter. Often class or object theory can become boring because we are discussing concepts that are at a higher level than code and data. Admittedly the theory is great, but there is nothing as rewarding as implementing the grand design, getting down to the code that returns results.

Designs on Classes

In Chapter 9, we looked at the key differences between inheritance, association, aggregation, and composition; here we will investigate some advanced patterns that give us the collateral needed for richer analysis, design, and depth. I have handpicked the following class design patterns because they have become the formative patterns in OO and are applicable in the .NET Framework. I discuss other patterns in subsequent chapters.

- **Singleton Pattern** The creational pattern that describes how to ensure that only one instance of a class can be activated. All objects that use the singleton instance use the same one.
- **Bridge Pattern** The structural pattern that prescribes the de−coupling of the implementation of a class from its interface so that the two can evolve independently of each other.
- **Strategy Pattern** The structural pattern that prescribes the de−coupling of the implementation of a class from its interface so that algorithms, or any operation or process, can be interchanged. Algorithm implementation can thus vary independently from the client or consumer objects that need it. Strategy is very similar to Bridge; however, Strategy is used to interchange implementation at runtime.
- **State Pattern** The behavioral pattern that provides a framework for using an object hierarchy as a state machineas an OO alternative to constant or variable state data, complex conditional statements, enumeration constants, and map tables.
- **Composite Pattern** The structural pattern that prescribes how classes can be composed (and aggregated) into tree−like hierarchies.
- **Iterator Pattern** The behavioral pattern for a class that provides a way to access the aggregated or composite elements of objects of a collection (as mentioned in Chapter 12, Microsoft's name for its iterator−like object is the "enumerator"). This chapter will implement an iterator as an implementation of The **IEnumerator** interface as described in the previous chapter (see the "IEnumerator and IEnumerable" section).
- **Adapter Pattern** The structural pattern that prescribes the conversion of an interface to a class into another interface a client object can use transparently. Adapter, affectionately known as Wrapper, is discussed at length in Chapter 14.
- **Null Pattern** A behavioral pattern providing an alternative to **Nothing** in Visual Basic .NET (see Chapter 14, "Iterating over a Tree").
- **Delegate Pattern** The Delegate pattern prescribes how to wrap a singleton method signature in a

Connections have to be managed according to the state they are in at any given time. A database or network connection may be open, closed, established, listening, receiving, or disconnecting.

Telecommunications and telephony applications are huge state machines that schedule operations according to the state of many different variables. Such applications are often called state machines. They are used so often in software solutions that they are part of the first year curriculum of every computer science course.

A PBX tests state and doles out operations accordingly. Lines and stations may be in busy state, off−hook state, on−hook state, ready state, out−of−service state, or logged−out state. These conditions are usually represented by enumeration constants.

In procedural programming worlds, state machines are typically managed in long and complex conditional elements such as **IfThen** and **Else If** constructs and **Case** and **Switch** blocks. But long conditional statements that test flags and values are undesirable in both procedure−oriented programming and Object−Oriented programming, because they make programs hard to read and maintain.

Mapping state transitions in a map table is an outmoded practice, although it is still usedeven in OO circles. If you recall, we reviewed something similar in the **GetMessages** example in Chapter 5and we used a map table in order to not preempt the discussion of classes and objects that began in Chapter 8.

Often programmers set up state machines without thinking ahead to how the application may change in the future. Thus, they must alter the machine frequently, and as they apply new conditions and operations, the entire design begins to unravel.

The State pattern lets us drop such conditionals and "lookup" constructs completely. This pattern permits us to create instances of state objects, or stateful objects, that derive from a hierarchy in which the concrete−state classes represent the states in the application. This hierarchy is illustrated in Figure 13−6.

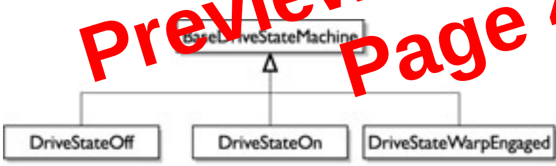


Figure 13−6: A hierarchy of state objects for a warp drive
The client object that needs to maintain the state machine does so by maintaining references to the state objects. Figure 13−7 shows how our **ShuttleInjector** object references objects of type **BaseDriveStateMachine** to determine the applicability of certain operations dependent on the state of the injector at any given time.

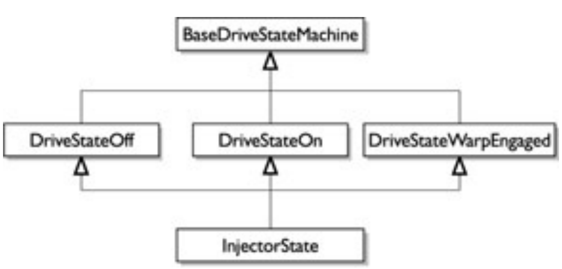
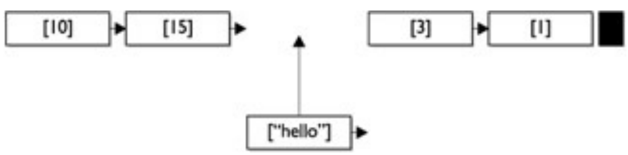


Figure 13−7: The class maintains the current state of the injector at all times by referencing a state object
For example, an injector cannot be placed into warp−ready state unless it has been started, and you would not want to try and start the warp drive if it has already been started. So the first state the injector would reference on the warp drive would be **DriveStateOff**. Our **BaseInjector** code can thus define the state−aware methods for all subclasses. For example, before the **Start** method of a **ShuttleInjector** object is called, our code should

In the illustration, the first node in the list, the tail, contains the **Integer** value 1. It represents the first item in the list. The last node, currently the *head node*, contains the **Integer** 10 and links to a node containing 15. Every linked list is built in this fashion. When we create the first node at the beginning (if no node precedes it), we mark the end with an end−of−list symbol. The letter "E" suffices, but in the actual code the link points to *null* (**Nothing**).

Visual Basic, or any other language for that matter, knows nothing about linkers and nodes in lists and trees. We represent these "concepts" programmatically using classes, objects, and data.

As mentioned earlier, linked lists operate like the standard stack or queue (see Chapter 12). The only difference is that you can insert new nodesthe dataanywhere in the structure of the linked list by maintaining references to *next* and *previous* nodes. How a node is referenced (last, first, current, previous, next, top, and so on) depends on the operation you need to perform relative to the current reference, the current position. From the perspective of the "current node" the node that was created after it is the next node and the node that was created before it is the previous node. This may be clearer in the illustration.



When you insert, only the neighboring nodes need to change. Inserting is similar to people jostling for positions in a lunch line. The illustration on the previous page shows this "pushing in" activity.

Removing a node follows the same pattern. You affect only the neighboring links and need to manage the references on both sides so that you can "plug" the hole that results upon the removal of a node.

Perhaps you are wondering why this is significant, since you can do this with arrays. Furthermore, arrays let you access an element or subscript value anywhere in the structure by virtue of the index. The specialty of the linked list, however, is that insertion, removal, and iteration are much faster and less resource−intensive than array insertion and removal, which indexes the elements for random accessing, sorting, and searching. Linked lists don't have the overhead associated with managing indexes. You gain speed but lose the benefit of random access. If you index the linked list, you are only steps away from concocting a custom array.

You cannot simply access any node in a list as you can an indexed element of an array. To remove or insert a node at a certain position in the list, you need to iterate through the entire sequence, one node at a time. This scrolling activity is very fast and used mostly to display or print the listand feed data to an array or other data structure or a stream (see Chapter 15).

Note If you are going to implement linked lists (and trees for that matter), and you expect them to grow big, don't formally index the structure. Linked lists are typically used for algorithms that don't need random access to elements. You typically process the roster as a unit. If you need a structure that gives you random access via an indexed element, use an array (see Chapter 12).

Looking at our examples of lists and trees, we realize that we don't need to sort the items. Furthermore, sorting would violate the integrity of the list. The list of Web sites recently visited, for example, would be worthless if you decided to sort it. On the other hand, the nodes of a list are easily accessed, so sorting them would not be very difficult. It is also quite easy to transfer the data to an array (as we will discover later in this chapter) and back again to a list.

```
      LastNode = TopNode
    End If
    num += 1
    Return num
  End With
End Function
```

The **num** variable plays an important role in the **Count** property, which is used in several places to report on the number of nodes in the list. **Count** is implemented shortly and the role of **num** will become more apparent.

The following code shows how to use **Add**:

```
Dim CopyList As New BaseNodeCollection()
Dim BigNode As BaseNodeCollection.Node
For Each BigNode In List
  CopyList.Add(BigNode)
Next BigNode
```

The Clear Method

The **IList.Clear** method clears all nodes from the list. This is a very easy method to implement. We simply de−reference the last node and the entire list collapses like a stack of cards. The garbage collector will conclude that there is no longer an interest in the entire chain and will collect all the objects accordingly.

The definition for the **Clear** method is as follows:

- **Method Name: Clear** The method clears the reference to a list in **BaseNodeCollection**
- **Method Signature** The method takes no arguments.

  ```
  Public Sub Clear() Implements IList.Clear
  ```
- **Parameters** None
- **Precondition** None; simply assigning **LastNode** to **Nothing** severs the connection to the list
- **Postcondition** None
- **Exceptions** None; this method will not throw an exception, even if the list does not exist

An implementation of the **Clear** method is as shown in the following code:

```
Public Sub Clear() Implements IList.Clear
  LastNode = Nothing
End Sub
```

The **Clear** method can be called as follows.

```
List.Clear()
```

The Contains Method

The **IList.Contains** checks to see if the specified value is contained in the **Data** object of the **Node**.

The definition for the **Contains** method is as follows:

- **Method Name: Contains** The method checks for the first existence of the specified value and

- **Exceptions** This method throws an exception of type **ArgumentOutOfRangeException** when the index specified does not exist in the list (it is thus outside the bounds of the structurethat is, below zero and higher than **Count**). The **Catch** handler also writes the exception's message to the **exceptInfo** field which is scoped to **BaseNodeCollection**.

An implementation of the **FindItem** method is as shown in the following code:

```
Private Sub FindItem(ByVal nodeIndex As Integer)
  Try
    If (nodeIndex < 0 Or nodeIndex > Count) Then
      Throw New ArgumentOutOfRangeException()
    End If
      Dim myIterator As System.Collections.IEnumerator = _
        Me.GetEnumerator()
      Dim intI As Integer = -1
      While intI < nodeIndex
        myIterator.MoveNext()
        CurrentNode = myIterator.Current()
        intI += 1
      End While
  Catch Except As ArgumentOutOfRangeException
    exceptinfo = Except.Message
  End Try
End Sub
```

This method requires a little more explanation. If the **nodeIndex** parameter is valid the first step required is to bridge an iterator object to the list. This is achieved in the following lines of code:

```
Dim myIterator As System.Collections.IEnumerator = _
  Me.GetEnumerator()
```


The nifty thing about this code is that it uses the **Me** "handle" to send a message to the current **BaseNodeCollection** object's **GetEnumerator** method. **GetEnumerator** is implemented to support instantiating the **IEnumerator** object. (The discussion of the **GetEnumerator** is coming up next.)

The next job of the method is easy. Once we have the iterator (of type **IEnumerator**) we can simply enter the list at the head, and iterate over the list until we arrive at the *n*th node specified by the **nodeIndex** parameter.

The iterator itself has a handle on the node it lands on via the **IEnumerator**'s current method (you will see how this works when we tackle the implementation of the iterator object with **IEnumerator** later in this chapter).

The GetEnumerator Method

As mentioned, we will implement an iterator that supports the .NET **IEnumerator** interface towards the end of this chapter, so this discussion of the **IEnumerable.GetEnumerator** method may seem a little premature. However, **GetEnumerator** is required by all methods that seek a handle to the list in order to traverse it (and do other things to it). We use an iterator extensively from within **BaseNodeCollection** (allowing the container to work on its own list) but we also need to implement it to support the likes of external constructs, such as **For Each . . . Next,** which will not work without a **GetEnumerator** implemented in the target object. **For Each . . . Next**, as discussed in Chapter 6, essentially uses our custom iterator to loop up the list, but it needs **GetEnumerator** to bridge to the iterator. Later on in this chapter you will see how this all comes together like peas in a pod.

```
        Return True
      Else
        If (workList.CurrentPosition.NodeNext Is Nothing) Then
          Reset()
          Return False
        End If
          workList.PreviousNode = workList.CurrentPosition
          workList.CurrentPosition =_
          workList.CurrentPosition.NodeNext
          Return True
      End If
    Catch NExcept As NodeNotFoundException
      iteratorInfo = "No nodes exist in this container."
    End Try
  End Function

  Public ReadOnly Property Current() _
    As Object Implements IEnumerator.Current
    Get
      Return workList.CurrentPosition
    End Get
  End Property
End Class
```

The **MoveNext** method is the workhorse of this class. With its reference to an instance of **BaseNodeCollection**, which it receives upon instantiation via its **New** constructor, it traverses the list by shuffling the nodes into different positionsthe previous node is assigned to the current position and the current node is assigned to the next position and so on.

The **Reset** method is implemented very simply. It just causes the iterator to lose its place in the list. The next time you make a call to **MoveNext**, the iterator is forced to start from the beginning again. The **IEnumerator** interface specifies that **IEnumerator** objects typically scroll in one direction. The iterator shown here starts at the head of the list and proceeds to the tail, going from the last node that was added to the list to the first node that was addedas if the list of nodes is a stack. The current version of the iterator does not support backward scrolling.

Reset is also called in the constructor so that the iterator is automatically reset whenever **New** is called.

The last member implemented here is the **Current** property. It simply returns the **Node** object assigned to the **CurrentPosition**. Note that **CurrentPosition** and **CurrentNode** both refer to the same thing, only **CurrentPosition** is the **BaseNodeCollection** property that accesses the data from the internal and private **CurrentNode** variable.

Note The formal Iterator pattern specifies a **CurrentItem** method as well as a **Next** method that is the equivalent of **MoveNext**. It also supports indexing, which can be easily implemented but is not really a necessity.

The following code demonstrates the iterator at work. The method **PrintNodesDemo1** makes an iterator using the **BaseNodeCollection**'s **GetEnumerator** method, while **PrintNodeDemo2** does the same thing using the **For Each . . . Next** construct.

```
Module LinkedListDemo

  Dim List As New BaseNodeCollection()

  Sub Main()
```

```
    List.Add("I")
    List.Add("just")
    List.Add("love")
    List.Add("OOP")
    List.Add("with")
    List.Add("VB.NET")
    PrintNodesDemo1()
    PrintNodesDemo2()
  End Sub

  Public Sub PrintNodesDemo1()
    Dim myIterator As System.Collections.IEnumerator = _
    List.GetEnumerator()
    While myIterator.MoveNext()
       Console.WriteLine(myIterator.Current.Data.ToString)
    End While
  End Sub

  Public Sub PrintNodesDemo2()
    Dim element As BaseNodeCollection.Node
    For Each element In List
      Console.WriteLine(element.Data)
    Next
  End Sub
End Module
```

The printout to the console for both cases shown in the code is as follows:

```
I
just
love
OOP
with
VB.NET
```


Note The code for the **BaseNodeCollection**, **Iterator**, and **Node** classes can be found in the Nodals project in the Vb7cr solution.

Observations

This chapter extended our discussion of data structures and provided us with some interesting code. But most of all, it showed what's possible with a pure object−oriented language like Visual Basic .NET. We saw many scenarios creating linked lists wherein objects and their interfaces are aggregated into container classes after they have been first defined as composite classes. We also looked at how Visual Basic can adoptand then run withmany of the formal patterns that have emerged to assist OO design and development over the years. Some of these patterns could be represented with classic VB. However, it is the native support for interfaces, polymorphism, encapsulation, and inheritance that makes all of the patterns adopted by languages such as Java and C++ more than applicable to Visual Basic .NET and the .NET Framework.

We are going to take this further in the next chapter, where we'll look at patterns for adapting interfaces, delegations, and delegates, as well as some advanced uses of interfaces.

32−bit pointer to the character data) that the COM object understands. **BSTR**s are thus converted to **String**s when the data comes back from the COM world. **String**−like data usually requires conversion while other types, such as a 4−byte **Integer**, require none.

The classes that wrap COM objects expose the COM interfaces to the .NET clients transparently and allow the COM objects to access components as if they were .NET objects. Wrapping takes into account all the HRESULTS, return values, reference counting, and other COM ingredients.

In the next chapter I examine another "wrap"the File System Object for files and folders (otherwise known as **FSO**), and the Index Server COM object. These COM objects were cooked up long before the .NET Framework arrived on the menu of development options, yet they partner well with .NET. So, if you have any investment in unmanaged code exposed as COM objects, they are automatically available to .NET clients.

If you have an investment in unmanaged code that you want to expose to .NET, and the code is not exposed as COM objects, then you have three choices. First, you could rewrite your code in Visual Basic .NET, which would probably be too time−consuming and expensive. Second, you could create a new custom interop layer for your code, an alternative that is less expensive than the first option but still a complex undertaking. Third, you could create the necessary COM−type libraries for the code (with a tool like Visual J++). The latter would require the least effort and expense, and it is preferable to expose the unmanaged code with COM interfaces rather than rewrite it for .NET.

Note We must remember that adding interoperability impacts performance, no matter how unnoticeable it may be. It is best to try to work with the classes in the .NET base class library and leave COM interop to your "out−of−options" situations, if only to get used to using the native classes.

Taking unmanaged code interface adaptation further is beyond the scope of this book. However, we do need to determine how to adapt classes within our operating framework. In other words, let's first ascertain what it means to adapt .NET interfaces for use by other .NET classes and objects. This will put us on the road to understanding **Delegates** and events.

The Adapter Pattern in .NET

The Adapter pattern prescribes how an **Adapter** class or object adapts an interface that clients will be able to use and couples it with a **Receiver**'s interface that the clients do not know how to use. For the record, the original implementation in the **Receiver** does not need to be known by the clients and it can varywhich is polymorphism in all its magnificent glory (see the related Bridge and Strategy patterns in the last chapter).

Objects and classes can receive messages either directly or indirectly. The following illustration first shows the normal process of sending the call message directly to an object with which it knows how to communicateby direct reference to a class or an object.

When a client object needs to call a method in the server objectbut it cannot call the method directly, as it normally would in an association or instantiation context between the two objectsit makes the call by way of an **Adapter** or a proxy. The message may be sent to the **Adaptee** or **Delegate**, which provides an interface. In Figure 14−5, the **Sender** sends a method call to an interface and has no knowledge, or desire to have

```
  End Class 'TrajectoryAdapter
End Class 'Trajectory
```

The Interface contains the singleton method reference.

```
Public Interface IRocLoc
  Function RetrRocLoc() As Coordinates
End Interface
```

And the **Sender** stays the same.

```
Inports Vb7cr.Trajectory
Public Class TrajectoryConsole
  Dim FindRoc As TrajectoryAdapter
  Dim GetRoc As IRocLoc = FindRoc
  Public Sub ObtainRocHeading()
    Plot(GetRoc.RetrRocLoc())
  End Sub
End Class
```

What does the **Adapter** pattern achieve?

- The **Sender** and the **Receiver** remain completely disinterested in each other's existence. It's not a matter of loose coupling; there is no coupling at all because the **Sender** has no way of accessing the private data and methods of the **Receiver**. In the above example, the **TrajectoryConsole** makes a reference to the **Adapter,** which it delegates to. If the **Adapter** implements an **Adaptee** interface then the de−coupling becomes more radical because only the implemented method of the **Adaptee** can be called at the **Adapter**. In both cases the **Adapter** makes a private privileged call to the **Receiver**, where the ultimate implementation lies, which handles the call.
- Method indirection. The "contra−indication" for this loose coupling scenario is that more complexity is added to the application, and it thus becomes a lot more difficult to understand. So all good adaptation needs to be accompanied by clear documentation and diagrams.
- The ability to use an existing class or object whose interface is not suitable for the clientsuch as referencing COM from .NET applications.
- The ability to create a class that can be used by a wide number of clients local to the framework and even foreign to it. Providing good interface, **Adaptee** support, and pluggable interfaces will help your class become as widely distributed as possible.
- The ability to adapt the original interface of a parent through the multiple implementation of more than one interface. This lets you use any existing subclasses of the parent without having to create an adapter for each subclass.
- The ability to provide an event model in which one or multiple **Receiver** objects, given the alias of **Listener**, can receive the event communications initiated at the **Sender**.

The consequences of adapting an object differ from those of adapting a class. A single **Adapter** object can be engineered to collaborate with many **Adapter** objects, including the other **Adapter**s of subclasses of the parent **Receiver**.

The downside of adapting the object rather than the class is that you lose the ability to override easily. In order to override the **Receiver**'s methods, you will need to create a child−class of the **Receiver** and make this derived/composite class the **Adapter** instead. This also works around the issue of having to share (make static) the method in the **Receiver**, which may not always be convenient or desirable.

parameter represents the type of **Delegate** to create, the **Object** parameter represents the class instance on which the method is invoked, and the **String** parameter represents the name of the instance method that the **Delegate** is to represent.

- **CreateDelegate(Type, Type, String)** This method creates a **Delegate** of the specified type that represents a static method in a specified class. The first **Type** parameter represents the type of **Delegate** to create, and the second **Type** parameter represents the type representing the class that implements the method. The **String** parameter represents the name of the static method that the **Delegate** is to represent.
- **CreateDelegate(Type, Object, String, Boolean)** This method creates a **Delegate** of the specified type that represents the specified instance method to invoke on the specified class instance with the specified case−sensitivity. The **Type** parameter represents the type of **Delegate** to create, the **Object** parameter represents the **Receiver** class instance on which method is invoked, the **String** parameter represents the name of the instance method that the **Delegate** references, and the **Boolean** parameter represents **True** or **False,** indicating whether to ignore the case when comparing the name of the method.

The following version of the **TrajectoryConsole** application makes use of the late bound semantics. First we cook up the **Delegate** as we did before.

```
Delegate Function GetRocLoc(ByVal some As Integer) As Coordinates
```

Then we set up the late bound declarations in the **Sender** object as shown in the following code.


```
Public Class TrajectorConsole

  Dim Traj As New Trajectory()
  Dim GetRocDel As GetRocLoc
  Dim Traj As New Trajectory()

  Public Sub ObtainRocHeading(ByVal opt As Integer)
    Try
      Select Case Option
        Case 0
          GetRocDel = CType(CreateDelegate(GetType(GetRocLoc), _
          Traj, "CurrentRocLoc"), GetRocLoc)
          GetRocDel.Invoke(Asteroids.AlphaAsteroid)
        Case 1
          GetRocDel = CType(CreateDelegate(GetType(GetRocLoc), _
          Traj, "AltCurrentRocLoc"), GetRocLoc)
          GetRocDel.Invoke(Asteroids.BravoAsteroid)
        Case 2
          GetRocDel = CType(CreateDelegate(GetType(GetRocLoc), _
          Traj, "PortCurrentRocLoc"), GetRocLoc)
          GetRocDel.Invoke(Asteroid.CharlieAsteroid)
        Case Else
          GetRocDel = CType(CreateDelegate(GetType(GetRocLoc), _
          Traj, "StarboardCurrentRocLoc"), GetRocLoc)
          GetRocDel.Invoke(Asteroids.ZuluAsteroid)
       End Select
    End Try
  End Sub
End Class
```

The utility you get from the late declares is evident in the example here where a **Select Case** statement block is used to upcast the **Delegate** variable at exactly the time it is needed. What's the beef? As you can see you can vary which method gets called in the **Trajectory** object.

method sans the recursion:

```
Public Overloads BubbleSort(ByRef array() As Integer, _
  ByVal outerStart As Integer, _
  ByVal innerStart As Integer, _
  ByVal bound As Integer)
  Dim outer, inner As Integer
  For outer = outerStart To bound
    For inner = innerStart To bound
      If (array(inner) > array(inner + 1)) Then
        Transpose(array, inner, inner + 1)
      End If
    Next inner
  Next outer
End Sub
```

There are no more recursive calls and no more stopping condition. But take note of the method that calls this **BubbleSort** method.

```
Public Class ArrayUtils

  Delegate Sub DoubleSortDel1(ByRef array() As Integer, _
    ByVal outer As Integer, _
    ByVal inner As Integer, _
    ByVal bound As Integer)
    Dim dblSort1 As DoubleSortDel1 = AddressOf Queuer.BubbleSort

  Public Sub PartitionSort(ByRef Array() As Integer)
    dblSort1(Array, 0, 0, Array.Length \ 2)
    dblSort1(Array, Array.Length \ 2 + 1, Array.Length \ 2 + 1, Array.Length - 2)
    Merge(mergearray)
  End Sub
End Class
```


The **PartitionSort** method almost concurrently sorts the two parts of the single array using the two **Delegate**s. The first **Delegate** sorts the first half and the second **Delegate** sorts the second half. Lastly, the independent call to the **Merge** method combines the partitions into one array.

The **QuickSort** method has a lot more potential for implementing **Delegate**s. First, the **QuickSort** with recursive calls and areas can be replaced with delegate calls called out in bold:

```
Public Overloads Sub QuickSort(ByRef Array() As Integer, _
  ByVal outerStart As Integer, _
  ByVal innerStart As Integer, _
  ByVal bound As Integer)
  Dim outer, inner As Integer
  If Not (outerStart >= Array.Length - 1) Then
    If (bound <= 0) Then
      bound = QuickPart(Array)
    End If
    For outer = outerStart To bound
      For inner = innerStart To bound
        If (Array(inner).CompareTo(Array(inner + 1))) > 0 Then
          Transpose(Array, inner, inner + 1)
        End If
      Next inner
    Next outer
      QuickSort(Array, outer, inner, Array.Length - 2)
```

regarding the coordinates of the asteroid and can pass this information to the **Delegate**. The **Delegate** then invokes the method in the weapons systems represented by the **Weapons** class and fires a laser at the approaching asteroid. To cater to this algorithm, the application could expose an **AsteroidEnter** event or the applications could simply invoke the **Delegate** object.

The class that encapsulates the events maintains the current state of the application, possibly by implementing a state machine, as discussed in the previous chapter. The states provide key information about each event and the operating mode of the application. So in "scanning" or "sensing" mode the application watches for that pesky asteroid and as soon as the closest one returns threatening data the event is fired. The following code is doing exactly what I have just described:

```
Public Class Trajectory

  Dim TrajState As New TrajectoryState
  Dim aSensors As New AsteroidSensors()

  Public Function CurrentRocLoc(ByVal ast As Asteroids) As Coordinates
    CurrentRocLoc = aSensors.RetrieveAlpha()
    Return CurrentRocLoc
  End Function

  ReadOnly Property IsEnabled() As Boolean
    Get
      Return TrajState.CurrentState
    End Get
  End Property
End Class

Public Class WeaponsArray

    Public Sub FireAsteroidLaser(ByVal roc As Asteroid, _
      ByVal loc As Coordinates)
      'code not implemented until laser gun meets
       universal standards
      Console.WriteLine("Firing Laser")
    End Sub

End Class

Delegate Function GetRocLoc(ByVal roc As Asteroids) As Coordinates

Public Module TrajectorConsole
  Dim Traj As New Trajectory()
  Dim Weps As New WeaponsArray()
  Dim RocLoc As New Coordinates()
  Dim Roc As New Asteroid()

  Dim GetRocDel As GetRocLoc = AddressOf Traj.CurrentRocLoc
  Delegate Sub FireLaser(ByVal roc As Asteroid, ByVal loc As Coordinates)
  Dim FireIt As FireLaser = AddressOf Weps.FireAsteroidLaser
  Public Event AsteroidEnter As FireLaser

  Public Function ObtainRocHeading() As Coordinates
    RocLoc = GetRocDel(Asteroids.AlphaAsteroid)
  End Function  Public Sub WatchForRock()
    While Traj.IsEnabled
      ObtainRocHeading()
        If Not (RocLoc.X And RocLoc.Y) > AlertEnum.StandDown Then
          'Or FireIt(Roc, RocLoc)
```

Chapter 15: Data Processing and I/O

Overview

This chapter deals with Visual Basic .NET's and the .NET Framework's text, character, and binary data processing abilities, as well as the I/O support for streams. We also introduce the regular expression classes and file operations and get acquainted with the extensive support for XML. Data processing and I/O represents the largest chapter in this book (and in most programming books) because it represents the most common task any programmer will be required to perform, from simply reading a command−line argument to loading a data warehouse with a hundred million bytes of information.

The discussion of files and streams also provides extensive examples of managing files and folders, streaming data to and from objects (serialization), and more. Much of the code examples were extracted from a utility called Indexworks, which tests classes I built to work against Microsoft Index Server. These include examples that write noise words (words to strip out of search phrases that Web surfers submit) to a noise words file that is loaded into an array or a linked list in the objects that send queries to Index Server.

The last section in this chapter "Serialization with XML" demonstrates providing XML serialization support for the linked list and node objects we worked on in Chapter 13. It follows after a long discussion on file I/O and demonstrates how to serialize the entire linked list and its node out to a file on the hard disk. It will show how, when starting the application, the entire linked list object and its data nodes can be reconstituted back into the application for immediate use.

Data Processing

Many languages are judged by their ability to manipulate and manage text and characters. Visual Basic .NET is no exception. The reason is simple: Without this fundamental ability, we would be unable to process data and represent it to our users, store it in databases, or print it to documents. There is hardly an application or algorithm that does not require the facilities for some form of text or character manipulation. We write text to the console, to dialog boxes, to event logs, and to the Debug Output window. We capture text from user input, such as reading a character from the console. We break text apart, interpret it, clean it up, and send it back to the screen, to databases, to files, to printers, to e−mail and pagers, and to remote devices.

In today's highly distributed world, text is king. The days of jumping through hoops and eating fire to get binary objects from one point on a network to another have been put behind us with the advent of XML, a sophisticated metadata framework for describing individual elements represented as text. Nowadays, all forms of data, including data destined only for computer consumption, travels with the elements that describe it. This so−called metadata, couched in XML tags, has turned text into the universal language of computing. As long as a receiver can read the XML (using an XML parser or method that reads XML tags) and can support what the text requires, it will know what to do with it.

In the not too distant past, sending a simple string from a VB application to a Java application or a Delphi application (or vice versa) was akin to cracking a coconut with a crayon. Each language would encode and encapsulate its text in a form that other applications could not easily translate. Strings wrapped in various codes needed to be unwrapped or translateda process akin to the translation of English between a Mississippi maiden out on a first date with a soccer freak from Liverpool. XML, the universal translator, changes all that.

The .NET Framework provides the power of text and character manipulation and processing in the form of several classes that have an exceptional assortment of features for you to use. In particular, we will look at the

CopyTo

The **CopyTo** method is a little more complex than the **Copy** method, but it works harder to give much more manipulation power. The **CopyTo** method takes a character at the source position of a **String**, at your selected index value, and then copies the character to a destination position in a character **Array**. The base syntax is as follows:

```
Str1.CopyTo(int1, myArray, int2, int3)
```

The character at **int1** is the starting point or source index in the source **String**in the preceding example, the source is **Str1**. The parameter **myArray** is the destination **Array** you must provide. Finally, **Int2** is the starting index or destination index in the target **Array** and **Int3** is the number of characters to copy from the source **String**, as shown in the following example:

```
Dim intI As Integer
Dim str1 As String = "Houston, we have a problem."
Dim myArray(5) As Char
str1.CopyTo(0, myArray, 0, 4)
str1.CopyTo(10, myArray, 4, 1)
For intI = 0 To 4
  Console.WriteLine(myArray(intI))
Next I
```

In the preceding code example, we have declared an array (**myArray**) of type **Char** to hold five characters. Then we copy four characters into **myArray** starting at index 0 and ending at index 3. Next, using **str1**, we copy character "e" in position 10 in the **String** to the index position 5 in the **Array**. The characters copied into the **Array** are "h," "o," "u," "s," and "e."

Finally, to write the **Array** contents to the console, we used a **For . . . Next** loop (refer to Chapter 6), which loops four times to output the characters and display the following:

```
h
o
u
s
e
```

EndsWith, StartsWith

The **EndsWith** and **StartsWith** methods are useful for simple checks on whether certain **String**s or even single characters appear at the beginning or end of **String**s. You will receive **True** or **False** if the **String** you are hoping to find *is* or *is not* at the end or beginning of your **String.** Let's check out this useful method:

```
Str.EndsWith()
Str.StartsWith()
```

Have a look at the following example:

```
Dim str1 As String = "Houston, we have a problem."
If str1.EndsWith("problem") Then
  console.WriteLine("true")
End If
```

IndexOf, LastIndexOf

The **IndexOf** and **LastIndexOf** methods provide a facility for locating a character or a set of characters in a **String** object. In a word processing application, for example, you will want to provide your users with the facility of searching for and replacing strings. Consider the following code snippet:

```
Dim str1 As String
str1 = "I waste a lot of time playing with my xbox."
Console.WriteLine(str1.IndexOf("x"))
```

It is rather easy to work out in your head the output to the console. It is the integer 38 of coursebeing the last character in the above **String** object. If the character is not present in the **String**, a return value of 1 is reported.

The method **LastIndexOf** provides a slightly different facility. It reports the last occurrence of a particular character in the **String**. In the preceding example, there are two occurrences of "x" so the return value is 41. But if we searched for "o" we could get 17 as the return value because there are two occurrences of "o" in the **String**, and we are looking for the last one.

Insert

The **Insert** method inserts a **String** into another **String** in a location specified in the method. Consider the following code:

```
Dim str1 As String
str1 = "The little black xbox"
Console.WriteLine(str1.Insert(4, "very expensive "))
```


The **String** argument "very expensive" is inserted at integer 4 in the **String s1** to display to the console the following:

```
The very expensive little black xbox
```

Intern, IsInterned

Often, **String** objects can get quite large, and the task of comparing them can become quite slow in computing terms. The **Intern** method provides a facility for obtaining a reference to a **String** that speeds up comparison operations by an order of magnitude. The **Intern** method is also useful for creating **Strings** on−the−fly and then providing an immediate facility for using the **String** in a number of operations.

When you invoke the **Intern** method of different **String** objects that have the same content as the original **String** object, you will obtain a reference to the first object. For every object instantiated that is the same as the original object, you will obtain multiple references to the same object by interning each new **String** object. Interned **Strings** can be compared with the = operator (equals) instead of calling the more resource−intensive **equals** operator, which literally has to compare each character in the corresponding **String**.

The following code demonstrates the interning of **String** objects:

```
Public Sub TestIntern
Dim s1, s2, s3, s4 As String
s1 = "The small brown fox"
s2 = "The small brown fox"
```

Reformats **String**s	**Format**, **FormatCurrency**, **FormatDateTime**, **FormatNumber**, **FormatPercent**
Retrieves the sub−**String** the specified number of characters from the left or right	**Left**, **Right**
Retrieves a **String** left− or right−aligned to a specified number of characters	**LSet**, **RSet**
Retrieves sub−**String**s	**Mid**
Strips spaces from **String**s	**LTrim**, **RTrim**, **Trim**
Finds a sub−**String** in a **String** **InStr**, **InStrRev**	
Retrieves the **Integer** values associated with ANSI and ASCII characters	**Asc**, **AscW**
Retrieves the character associated with the specified character code	**Chr**, **ChrW**
Returns a **Char** value representing the character from the specified index in the supplied **String**	**GetChar**
Replaces one **String** with another	**Replace**
Retrieves subsets (as arrays) of **String**s from a filter applied to an array	**Filter**
Retrieves an array containing the result of splitting a **String**	**Split**
Retrieves the result of a join of two **String**s	**Join**

These **String** manipulation functions are just as useful in Visual Basic .NET as they are in classic VB. If you can easily solve your problems using the native **String** manipulation methods, then you should prefer those so that you lessen the burden of and reliance on legacy code. On the other hand, if one of these functions does the job, don't hesitate to use it. I have used several of these functions in .NET applications to reduce the amount of code I needed to write to achieve a certain result and I found no noticeable problems or overhead.

To use these functions, you need to reference the Visual Basic Run−Time Library.

String Formatting

As mentioned earlier, the .NET Framework provides three types of format providers. These provide formatting of numeric **String**s, data and time **String**s, and **Enumeration String**s. These "formatters" are wired into the **ToString** methods of the fundamental data types that implement the **IFormattable** interface, such as **Int32** (**Integer**), **Int64** (**Long**), **Single**s, **Double**s, **DateTime**, **Enumerator**, and the like.

As demonstrated earlier in this chapter and in various places in this book, these formatters are also present in the workings of the **Console** and **String** classes and other classes, such as those in the **System.IO** namespace, that process text. See the "Format" section earlier in this chapter.

Classes that provide the formatter "masks" or "patterns, such as {00:00} and separator tokens and decimal point tokens, are known as format providers. These classes implement the **IFormatProvider** interface.

The format provider is typically passed to an overloaded **ToString** method as defined by the **IFormattable** interface. If no provider is passed, then the method can be coded to use a default format provider against the arguments processed to it. In such situations where no providers are passed, the formatting is implicit to the method, which obtains the mask and its tokens from one of the standard framework format providers. However, **ToString** methods typically implement **IFormattable** to provide the support in one of their overloaded variations (such as **Console.WriteLine**).

The key format providers that implement the **IFormatProvider** interface are listed as follows:

- **NumberFormatInfo** Formatting information for numeric data types
- **DateTimeFormatInfo** Formatting information for **DateTime** objects
- **CultureInfo** Formatting information for different cultures

In cases where formatting information is needed but no **IFormatProvider** is supplied, the **CultureInfo** object associated with the current thread is usually used.

NumberFormatInfo

The standard **NumberFormatInfo String** comprises a character that represents the format, such as *currency* or *decimal*, followed by digits that represent the precision. Table 15−4 lists the standard formats supported by the **Format** method.

Currency

The **Currency** formatter is used to convert the given numerical value to a currency value. The currency value can contain a locale−specific currency amount. The format information is determined by the current locale, but you can override this by passing in the **NumberFormatInfo** object as an argument. The default in the United States is, of course, USD. For example:

```
Console.WriteLine("{0:c}", 1250.99)
Console.WriteLine("{0:c}", −1250.99)
```

Tip The **Console.WriteLine** method automatically calls **String.Format** as demonstrated in the preceding and following examples (as does **ToString** if the argument can be formatted as defined by the **IFormattable** interface).

Table 15−4: The Built−in Formatters, or Format Providers, that Implement *IFormatProvider*

Format Specifier	Output
C, c	**Currency**
D, d	**Decimal**
E, e	**Exponential** (scientific)
F, f	**Fixed−point**
G, g	**General**
N, n	**Number**
R, r	**Roundtrip**. This format ensures that numbers converted to **String**s will get the same value when they are converted back to numbers.
X, x	**Hexadecimal**

The output to the console is the following:

```
$1,250.99
($1,250.99)
```

Decimal

The **Decimal** formatter can be used to convert the numerical value to an **Integer** value. For example:

```
Console.WriteLine("{0:D}", 125099)
```

writes 125099 to the console, but

```
Console.WriteLine("{0:D10}", 125099)
```

writes 00000125099 to the console, representing ten digits (five as passed by the parameter and five zeros for left−padding).

Exponential

The **Exponential** formatter (scientific) formats the value passed to the **String** in the form of

```
m.dddE+xxx
```

As indicated, the decimal point is always preceded by one digit. The number of decimal places is specified by the precision specifier (six places is the default). You can use the format specifier to determine the case of the "E" in the output, as illustrated in the following examples:

```
Console.WriteLine("{0:E}", 125.8)
Console.WriteLine("{0:E10}", 125.88)
Console.WriteLine("{0:E5}", 125.88))
```

This example writes the following to the console:


```
1.258800E+002
1.2588000000E+002
1.25880e+002
```

Fixed−Point

The **Fixed−Point** formatter is used to convert the value provided in the argument to a **String** and then specify the number of places after the decimal point to round the number. For example, the following code:

```
Console.WriteLine("{0:F}", 125.88)
Console.WriteLine("{0:F10}", 125.88)
Console.WriteLine("{0:F0}", 125.88)
```

provides this output:

```
125.88
125.8800000000
126
```

General

The **General** formatter is used to convert the **String** to a numerical value of either fixed− point format or scientific format. This is often used in calculator software to write to the format that provides a more compact representation. For example, the following code:

y en	US	April, 2001
y af	ZA	April 2001
L en	UZ	Unrecognized format specifier; a format exception is thrown

Digit or Zero for a Placeholder

The following code formats the output to the designated number of digits using a zero as the placeholder. If there are more placeholders than digits passed in the argument, the output is left−padded with the placeholder zeros. For example:

```
Console.WriteLine("{0:111}", 1234)
Console.WriteLine("{0:00}", 12)
Console.WriteLine("{0:0000}", 123)
Console.WriteLine("{0:0000}", 1234)
```

provides the following output

```
111
12
0123
1234
```

In the preceding output, the first line generates three of digit "1" because this placeholder is not recognized by the method and is thus simply copied to the output, and the number (1234) as the argument is ignored. The second line shows output limited to two digits. The third line shows output limited to four digits, but because we only provide a three−digit **String** as the argument, the number is left−padded with a zero. The fourth line shows four numbers formatted to a **String** of four digits.

Using a Digit or Pound for a Placeholder

The pound (or hash) character can be used as the digit or space placeholder. This placeholder works just like the zero except that a space or blank is inserted into the output if no digit is used in the specified position. For example:

```
Console.WriteLine("{0:####}", 123)
Console.WriteLine("{0:####}", 1234)
Console.WriteLine("{0:##}", 123456)
```

writes the following output to the console:

```
123
1234
123456
```

Custom Positioning of the Decimal Point

You can determine the position of the decimal point in a **String** of numerals by specifying the position of the period (.) character in the format **String**. You can also customize the character used as a decimal point in the **NumberFormatInfo** class. Here is an example:

```
Console.WriteLine("{0:####.000}", 123456.7)
Console.WriteLine("{0:##.000}", 12345.67)
Console.WriteLine("{0:#.000}", 1.234567)
```

The following code writes the following **String**s to the console:

```
123456.700
12345.670
1.235
```

Using the Group Separator

The group separator is a comma (,) and can be used to format large numbers to make them easier to read. You typically add the comma three places after the decimal point to specify a number such as 1,000.00 or higher. The character used as the specifier can also be customized in the **NumberFormatInfo** class. The following example illustrates placement of the group separator:

```
Console.WriteLine("{0:##,###}", 123456.7)
Console.WriteLine("{0:##,###,000.000}", 1234567.1234567)
Console.WriteLine("{0:#,#.000}", 1234567.1234567)
```

The output to console looks like this:

```
123,457
1,234,567.123
1,234,567.123
```

Using Percent Notation

You can use the percent (%) specifier to denote that a number be displayed as a percentage. The number will be multiplied by 100 before formatting. In the following example:

```
Console.WriteLine("{0:##.000%}", 123.45)
Console.WriteLine("{0:00%}", 0.123)
```

you get the following percentages displayed in the console:

```
12,345%
12%
```

Building Strings with StringBuilder

The efficiency of the **String** object as an immutable type has its downside. Every time you change the **String**, you create a new **String** object that requires its own memory location. If you need to repetitively work with a **String**, shaping it for a particular task, you have the additional overhead of the constant creation of new **String** objects every time you need to cut, add, and move characters around in the **String**.

When you need to constantly work with a **String**, such as an algorithm that takes UNC paths and converts them to HTML paths, or when you need a storage location to shove characters into, like a stack, then you need to turn to the **StringBuilder** class. This class can be found on the **System.Text.StringBuilder** namespace and allows you to keep working with a **String** of characters represented by the same objects for as long as it is needed. The great feature of the object is that you get to reference the collection of characters as a single **String**far less code than that "soda−fountain" **Stack** that requires extensive "popping."

Note In the BitShifters code in Chapter 5, we used the **StringBuilder** object, albeit in C# garb, as a place to stuff bits.

Regex

This simple example shows how to use the **Regex.Replace** method when a match is found in the target that is being parsed. The **StripNoise** method looks in a **String** for all instances of words provided in the array of samples and then deletes the matches from the target **String**:

```
Private Function StripNoise(ByVal sentence As String) As String
  While x <= UBound(noiseArray)
    sentence = Regex.Replace(sentence, noiseArray(x), "")
    x += 1
  End While
    Return sentence
End Function
```

The method then returns the new **String** minus the matched words. Trying to use one of the **String** manipulation methods or functions in an extensive block of text or a stream of characters would be extremely difficult, and in many cases not at all possible.

You would also use regular expressions to parse complex TCP/IP headers to locate and remedy malformed URLs. You know how complex some of these headers can be. If you need to look for misplaced periods, white spaces, illegal characters, duplication of @ ("at") symbols, and so on, nothing other than a regular expression can do the job for you. I also use it to "scrub" data and "prepare" complex search **String**s supplied against the likes of the Microsoft Index Server search engine, which chokes on a comma or a period or misplaced white space.

File, Stream, and Text IO Operations

The .NET Framework provides an impressive range of IO namespaces that contain dozens of classes used for writing, reading, and streaming all manner of text, characters, and binary data, as well as file, folder, and path support. Many of them, such as those represented by the **System.IO**, **System.Text**, and **System.XML** namespaces, let you code asynchronous and synchronous reading and writing of data to streams and files. All these namespaces are currently partitioned across the **mscorlib**, **System**, **System.Text**, and **System.XML** assemblies.

The files you work with in your programs are typically ordered collections of bytes, representing characters on a file system. Files are static; they squat on your hard disks like chickens hatching eggs. Streams, on the other hand, are continuous "rivers" of data, writing to and reading from various devices. Streams are constantly on the move.

Streams typically originate from files on devices like hard disks, CDs, and DVDs, and other devices for persistent storage such as tape drives and optical disks. Streaming data moves across processes and workspaces on your workstation, and between computers on the vast networks of the world. They move from persistent memory into volatile memory and back again in a constant ebb and flow of data. Eventually, streams of data get fed to printers for physical representation like annual reports or user manuals (after some time, the pages can be fashioned into paper airplanes and tossed out of windows or shredded when the FBI comes knocking).

The following is a list of the key sections we will discuss that facilitate basic I/O for the .NET Framework:

- **File and Directories** This section presents the classes that encapsulate the functionality of all known file and directory processing operations. Files are opened, closed, and manipulated using the classes

The File Class

The **File** class provides the .NET Framework's support for all manner of file handling. With this class, you can create, copy, delete, move, and open files and much more. **File** is a static or shared "operations" class and not a class for objects. You cannot instantiate **File** and it has no constructor. To use **File**, simply reference it as follows:

```
File.Copy("c:\doh.txt", "c:\ray.txt")
```

If you need to use file objects to do your file operations, use the **FileInfo** class, which contains instance methods that work like the methods of the **File** class but live in objects. Static methods, being shared, incur more security overhead, whereas instance methods do not always require security checks. Table 15−9 lists the static (s) members of **File**.

Note See "Basic File Class Operations" at the end of this section for a discussion of the methods most frequently used for standard file I/O operations.

You have the potential to raise exceptions if you provide malformed filenames and path information. The following examples of paths will get processed by the **File** methods, but you should consider using the **Path** class, discussed next, to help reduce path errors:

```
"c:\doh\ray.txt"
"c:\doh"
"doh\ray.txt" 'a relative path and file name
"\\doh\ray\" 'A UNC path for a server and share name.
```


Path

Programming against the file and directory classes is not rocket science, but the potential for problems in your code is increased because you need to pass complex arguments (such as the various mode and access constants discussed in the next section). One parameter that can be a minefield represents the path information argument you need to pass to the various methods of the file and directory classes.

Table 15−9: The Static Methods of *File*

Member	Purpose
AppendText	Bridges to a **StreamWriter** that appends UTF−8 encoded text to an existing file
Copy	Copies a source file to a new target file
Create	Creates a file on a given fully qualified path
CreateText	Creates or opens a new file for writing UTF−8 encoded text
Delete	Deletes the file on the given fully qualified path. An exception is not thrown if the specified file does not exist
Exists	Checks if a specific file exists
GetAttributes	Retrieves the **FileAttributes** on the file on a given fully qualified path
GetCreationTime	Retrieves the creation date and time of the specified file or directory
GetLastAccessTime	Retrieves the date and time that the file or directory was last accessed
GetLastWriteTime	Retrieves the date and time that the file or directory was last written to

```
  End Get
End Property
```

The **PathRoot** information returned is

```
C:\
```

The following example tests to see if a logical root exists in a path string. It returns **False** when the **FilePath** property passes "indexwork\noisefile.txt" to the **IsPathRooted** method.

```
Public ReadOnly Property CheckRooted() As Boolean
  Get
    Return PathChecker.IsPathRooted(FilePath)
  End Get
End Property
```

Remember that **Path** is not privy to exactly what's cooking on the hard disks or devices, volatile or built of silicone and metal. Just because a drive and file path check though the **Path**'s string gauntlet does not mean the actual drive, computer, and network actually exist at the time the path checks out.

File Enumerations

Among the parameters required by various methods for file operations are certain values that are represented by a collection of enumeration classes. These classes include constants for file access, synchronous or asynchronous processing, and file attributes. Table 15−11 lists the file enumeration classes.

Note These enumerations can apply to **FileInfo** and **FileStream** classes as well, so get used to them now.

FileAccess

Various file−handling methods require you to specify the level of file access enjoyed by the user or process accessing the file. The default file access level is full *read* and *write* capability on a file. A **FlagsAttribute** attribute decorates the class (refer to Chapter 8) so that the CLR can evaluate bitwise combinations of the members of the enumeration. Table 15−12 lists the three **FileAccess** attributes.

Here is an example that grants read−only access to a file. This allows it to be opened by the **File** operation while someone else is using the file, but only allows the other, latter users to read the file. They cannot write to it until they get the chance to open the file with write access, as demonstrated in the following code:

```
Dim noisefile As New FileStream(filePath, FileMode.Open, _
FileAccess.Read, FileShare.Read)
```

Table 15−11: File Enumeration Classes

Enumeration	Purpose
FileAccess	Read and write access to a file
FileShare	Level of access permitted for a file that is already in use
FileMode	Synchronous or asynchronous access to the file in use

Table 15−12: Constants for the *FileAccess* Attributes Parameter

LastAccessTime (p)	Gets or sets the time the current file or directory was last accessed
LastWriteTime (p)	Gets or sets the time when the current file or directory was last written to
Length (p)	Gets the size of the current file or directory
Name (p)	Gets the name of the file
AppendText	Creates a **StreamWriter** that appends text to the file represented by this instance of the **FileInfo**
CopyTo	Copies an existing file to a new file
Create	Creates a file
CreateText	Creates a **StreamWriter** that writes a new text file
Delete	Permanently deletes a file
MoveTo	Moves a specified file to a new location, providing the option to specify a new filename
Open	Opens a file with various read/write and sharing privileges
OpenRead	Creates a read−only **FileStream**
OpenText	Creates a **StreamReader** with UTF8 encoding that reads from an existing text file
OpenWrite	Creates a write−only **FileStream**
Refresh	Refreshes the state of the object

Apart from the semantic differences, the reduction in security checks, and a few additional members like **Refresh** and **Length**, this class provides the same operations on files as the **File** class. As I said, if you get more utility out of a file system object and prefer to stick with a .NET file handling class, then use **FileInfo** over the legacy FSO.

Also, the same exception raised for **File** problems applies to **FileInfo** problems, especially malformed paths and file information.

DirectoryInfo

Table 15−18 lists the methods and properties of the **DirectoryInfo** class. This class can be instantiated and its members are instance members. Instantiation gets you access to a useful collection of properties that provide information such as file extensions, parent directory names, root folders, and so on.

Table 15−18: The Instance Members of the *DirectoryInfo* Object

Member	Purpose
Attributes (p)	Retrieves or changes the **FileAttributes** of the current resource
CreationTime (p)	Retrieves or changes the creation time of the current resource
Exists (p)	Retrieves or changes a value indicating whether the directory exists
Extension (p)	Retrieves or changes the string representing the extension part of the file
FullName (p)	Retrieves the full path of the directory or file
LastAccessTime (p)	Retrieves or changes the time the current file or directory was last accessed
LastWriteTime (p)	Retrieves or changes the time when the current file or directory was last written to

NotifyFilter (p)	Retrieves or changes the type of changes to watch for
Path (p)	Retrieves or changes the path of the directory to watch
Site (p)	See **Component.Site** of the **Component** class
SynchronizingObject (p)	Retrieves or changes the object used to marshal the event handler calls issued as a result of a directory change
BeginInit	Begins the initialization of a **FileSystemWatcher** used on a form or used by another component. The initialization occurs at run time.
EndInit	Ends the initialization of a **FileSystemWatcher** used on a form or used by another component. The initialization occurs at run time.
Equals (inherited from **Object**)	Determines whether two **Object** instances are equal
WaitForChanged	A synchronous method that returns a structure that contains specific information on the change that occurred
Changed (e)	Occurs when a file or directory in the specified Path is changed
Created (e)	Occurs when a file or directory in the specified Path is created
Deleted (e)	Occurs when a file or directory in the specified Path is deleted
Disposed (e) (inherited from **Component**)	Adds an event handler to listen to the **Disposed** event on the component
Error (e)	Occurs when the internal buffer overflows
Renamed (e)	Occurs when a file or directory in the specified Path is renamed

The following example watches a folder for changes in files that will trigger the need to re−create the files and folder status report. It creates a **FileSystemWatcher** to watch the directory specified at run time. The component is set to watch for changes in **LastWrite** and **LastAccess** time, and the creation, deletion, or renaming of text files in the directory. If a file is changed, created, or deleted, the path to the file prints to the console. When a file is renamed, the old and new paths print to the console.

```
Public Sub Watching()
  'declare a watcher
  Dim Watcher As New FileSystemWatcher()
  'specify a path
  Watcher.Path = "c:\indexworks"
  'specify the notify filters
  Watcher.NotifyFilter = (NotifyFilters.LastAccess _
  Or NotifyFilters.LastWrite _
  Or NotifyFilters.FileName _
  Or NotifyFilters.DirectoryName)
  'the file to watch
  Watcher.Filter = "noisefile.txt"
  'Specify event handlers.
  AddHandler watcher.Changed, AddressOf OnChanged
  'Start
  Watcher.EnableRaisingEvents = True
End Sub

'implement the event handler.
Public Shared Sub OnChanged(ByVal source As Object, _
  ByVal eArgs As FileSystemEventArgs)
  'Reload the noisewords file after it has been changed
  Indexworks.Reload(FilePath)
End Sub
```

The following ingredients in this code should be noted.

NotifyFilters

Changes to watch for in a file or folder are specified by constants of the **NotifyFilters** enumeration and set to the **NotifyFilter** property. Like the constants of the file mode, access, and attributes enumerations, this enumeration has a **FlagsAttribute** attribute that allows a bitwise combination of its member values. In other words, you can combine constants to watch for more than one kind of change. For example, you can watch for changes in the size of a file *or* a folder, *and* for changes in security settings. The combination raises an event anytime there is a change in size or security settings of a file or folder.

Table 15−21 lists the constants of the **NotifyFilters** enumeration.

Table 15−21: Members of the *NotifyFilters* Enumeration

Member	Purpose
Attributes	Represents the attributes of the file or folder
CreationTime	Represents the time the file or folder was created
DirectoryName	Represents the name of the directory
FileName	Represents the name of the file
LastAccess	Represents the date the file or folder was last opened
LastWrite	Represents the date the file or folder last had anything written to it
Security	Represents the security settings of the file or folder
Size	Represents the size of the file or folder

Use these filter constants to specify the constant or constant combinations to watch. To watch for changes in all files, set the **Filter** property in the object to an empty string (""). If you are watching for a specific file, then set the **Filter** property to the filename as follows:


```
Watcher.Filter = "noisefile.txt"
```

To watch for changes in all text files, simply set the **Filter** property as follows:

```
Watcher.Filter = "*.txt"
```

By the way, hidden files are not ignored.

WatcherChangeTypes

Changes to watch for that may occur to a file or folder are specified by constants of the **WatcherChangeTypes** enumeration and set to the event hanlder. This enumeration also has a **FlagsAttribute** attribute decorating it that allows the CLR to reference bitwise combinations of its member values. Each of the **WatcherChangeTypes** constants is associated with an event in **FileSystemWatcher**. The constant members of this enumeration are listed in Table 15−22.

Table 15−22: The Constants of the *FileSystemWatcher* Enumeration

Member	Description
All	Fires on the creation, deletion, change, or renaming of a file or folder

- **FileStream** Bridges a **Stream** object to a file for synchronous and asynchronous read and write operations. This class can be derived from and instantiated.
- **MemoryStream** Creates a **Stream** in memory that can read a block of bytes from a current stream and write the data to a buffer. This class can be derived from and instantiated.
- **CrytoStream** Defines a **Stream** that bridges data objects to cryptographic services. This class can be derived from and instantiated.
- **NetworkStream** Defines a **Stream** that bridges data objects to network services. This class can be derived from and instantiated.

The preceding classes provide data streaming operations. This data may be persisted by bridging certain objects to various backing stores. However, the **Stream** objects do not necessarily need to be saved. Your objects might stream volatile information, resident only in memory. For algorithms not requiring backing stores and other output or input devices, you can simply use **MemoryStream** objects.

MemoryStream objects support access to a nonbuffered stream that encapsulates data directly accessible in memory. The object has no backing store and can thus be used as a temporary buffer. On the other hand, when you need to write data to the network, you'll use classes like **NetworkStream**. Your standard text or binary streams can also be managed using the **FileStream** class. **Stream** objects enable you to obtain random access to files through the use of a **Seek** method, discussed shortly.

Another **Stream** derivative you will find yourself using on many occasions is the **CryptoStream** class. We'll go over it briefly later in this chapter. This class is also not included in the **System.IO** namespace but has been added to the **System.Security.Cryptography** namespace.

BufferedStream objects provide a buffering bridge for other **Stream** objects, such as the **NetworkStream** object. The **BufferedStream** object stores the stream data in memory in a special byte cache, which cuts down on the number of calls the object needs to be made to the OS.

FileStream

FileStream objects can be used to implement all of the standard input, output, and error stream functionality. With these objects, you can read and write to file objects on the file system. With it you can also bridge to various file−related operating system handles, such as pipes, standard input, and standard output. The input and output of data is buffered to facilitate performance.

The **File** class, discussed earlier in this chapter, is typically used to create and bridge **FileStream** objects to files based on file paths and the standard input, standard output, and standard error devices. **MemoryStream** similarly bridges to a byte array.

The principal method in a **FileStream** object is **Seek**. It supports random access to files and allows the read/write position to be moved to any position within a file. The location is obtained using byte offset reference point parameters. The following code demonstrates the creation and opening of a file and the subsequent bridge to the **FileStream** object used to write to the file:

```
Dim aFile As New FileStream(source, IO.FileMode.Create)
```

In the preceding code, a file is opened, or created if it does not already exist, and information is appended to the end of the file. The contents of the file are then written to standard output for display.

Byte offsets are relative to a seek reference point. A seek reference point can be the beginning of the file, a position in the file, or the end of the file. The three **SeekOrigin** constructs are the properties of the

If you always read and write for sizes greater than the internal buffer size, then **BufferedStream** might not even allocate the internal buffer. **BufferedStream** also buffers reads and writes in a shared buffer. Usually, you do a series of reads or writes, but rarely alternate between reading and writing.

The following method example demonstrates the creation of a **BufferedStream** object bridged to the earlier declared standard **FileStream** object:

```
Public Sub AddNoises(ByVal source As String, _
ByVal noiseword As String)
  Dim fStream As New FileStream(source, FileMode.OpenOrCreate, _
    FileAccess.Write)
  Dim bStream As New BufferedStream(fStream)
  'Create a 'StreamWriter' to write the data into the file.
  Dim sWriter As New StreamWriter(bStream)
  sWriter.WriteLine(noiseword)
  ' Update the 'StreamWriter'.
  sWriter.Flush()
  ' Close the 'StreamWriter' and FileStream.
  sWriter.Close()
  fStream.Close()
End Sub
```

NetworkStream

A **NetworkStream** object provides the underlying stream of data for network access. **NetworkStream** implements the standard .NET Framework stream mechanism to send and receive data through network sockets. It also supports both synchronous and asynchronous access to the network data stream.

CryptoStream

The CLR also uses a streams model for reading and writing encrypted data. This service is provided by the **CryptoStream** object. Any cryptographic objects that implement **CryptoStream** can be chained together with any objects that implement **Stream**, so the streamed output from one object can be bridged into the input of another object. The intermediate result (the output from the first object) does not need to be stored separately.

MemoryStream

MemoryStream is no different from the previously mentioned streams except that volatile memory is used as the so−called backing store rather than a disk or network sockets. This class encapsulates data stored as an unsigned byte array that gets initialized upon the instantiation of the **MemoryStream** object. However, the array can also be created empty. The encapsulated data in the object is thus directly accessible in memory. **Memory** streams can reduce the need for temporary buffers and files in an application, which can improve performance by an order of magnitude.

GetBuffer

To create a **MemoryStream** object with a publicly visible buffer, simply call the default constructor. A stream can be declared resizable, which resizes the array, but in that respect, multiple calls to **GetBuffer** might not return the same array. You can also use the **Capacity** property, which retrieves or changes the number of bytes allocated to the stream. This ensures consistent results. **GetBuffer** also works when the **MemoryStream** is closed.

ToArray

The **ToArray** method is useful for translocating the contents of the **MemoryStream** to a formal **Byte** array. If the current object was instantiated on a **Byte** array, then a copy of the section of the array to which this instance has access is returned. **MemoryStream** also supports a **WriteTo** method that lets you write the entire contents of the memory stream to another streamone that has a persistent backing store, for example.

Readers and Writers

So far we have looked at classes that let you work with **String**s that also provide a facility for you to retrieve or supply the **String**. We have also gone from manipulating **String** data to constructing **String**s and using them in various display fields. While capturing the values provided by the various **ToString** methods is possible, the classes and utilities discussed earlier don't provide much in the way of features that get data on the road. **StringReader** and **StringWriter** provide the basic facilities for character I/O.

The .NET Framework's data streaming (I/O) support inherits from the abstract **TextReader** and **TextWriter** classes that live in the **System.IO** namespace. These classes form the basis of support for internationally viable and highly distributed software because they support Unicode character streams.

Text Encoding

Before we look at the reader and writer classes, understand that methods are provided to convert **Array**s and **String**s of Unicode characters to and from **Array**s of **Byte**s encoded for a target code page. A number of encoding implementations are thus provided in the **System.Text** namespace. The following list presents these encoding classes:

- **ASCIIEncoding class** Encodes Unicode characters as single 7−bit ASCII characters. It only supports character values between U+0000 and U+007F.
- **UnicodeEncoding** Encodes each Unicode character as two consecutive **Bytes**. Both little−endian (code page 1200) and big−endian (code page 1201) **Byte** orders are supported.
- **UTF7Encoding** Encodes Unicode characters using the UTF−7 encoding (UTF−7 stands for UCS Transformation Format, 7−bit form). This encoding supports all Unicode character values, and can also be accessed as code page 65000.
- **UTF8Encoding** Encodes Unicode characters using the UTF−8 encoding (UTF−8 stands for UCS Transformation Format, 8−bit form). This encoding supports all Unicode character values, and can also be accessed as code page 65001.

Other encoding can be accessed using the **GetEncoding** method that passes a code page or name argument.

When the data to be converted is only available in sequential blocks (such as data read from a stream), an application can use a decoder or an encoder to perform the conversion. This is also useful when the amount of data is so large that it needs to be divided into smaller blocks. Decoders and encoders are obtained using the **GetDecoder** and **GetEncoder** methods. An application can use the properties of this class, such as **ASCII**, **Default**, **Unicode**, **UTF7**, and **UTF8**, to obtain encodings. Applications can initialize new instances of encoding objects through the **ASCIIEncoding**, **UnicodeEncoding**, **UTF7Encoding**, and **UTF8Encoding** classes.

Through an encoding, the **GetBytes** method is used to convert arrays of **Unicode** characters to **Array**s of **Bytes**, and the **GetChars** method is used to convert **Array**s of **Byte**s to **Array**s of Unicode characters. The

ReadLine	Reads a line from the underlying string
ReadToEnd	Reads to the end of the text stream

Table 15−28: The Members of the *StringWriter* Class

Member	Purpose
Encoding (p)	Retrieves the encoding in which the output is written
FormatProvider (p)	Retrieves an object that controls formatting
NewLine (p)	Retrieves or changes the line terminator string used by the current **TextWriter**
Close	Closes the current **StringWriter** and the underlying stream
CreateObjRef	Creates an object that contains all the relevant information required to generate a proxy used to communicate with a remote object
Flush	Clears all buffers for the current writer and causes any buffered data to be written to the underlying device
GetLifetimeService	Retrieves the current lifetime service object that controls the lifetime policy for this instance
GetStringBuilder	Returns the underlying **StringBuilder** object
Write	Writes to this instance of the **StringWriter**
WriteLine	Writes some data as specified by the overloaded parameters, followed by a line terminator

Write

User the **Write** method to write data to an object that can receive a text input stream, such as **StringBuilder**. The following code illustrates writing to a **StringBuilder** object.

```
Public Sub AddWords(ByVal source As String, ByVal newword As String)
  Dim sBuilder As New StringBuilder()
  Dim strWriter As New StringWriter(sBuilder)
  strWriter.Write(newword)
  'check if write worked
  Console.WriteLine(sBuilder)
End Sub
```

Write does not force a new line, and new text is either appended to the existing text or, depending on the object, overwrites it.

WriteLine

The **WriteLine** method works exactly like **Write**, but adds a line terminator to the end of the **String**, which forces a new line. This is demonstrated as follows:

```
Console.WriteLine(sBuilder)
```

GetStringBuilder

The **GetStringBuilder** method will return an instance of **StringBuilder** that you can write to. Append and insert your characters to the object as demonstrated in the earlier "Building Strings with StringBuilder" section and then simply write the object to a line.

```
Public Sub AddChars()
```

The number is reset to 1 for each element.

Attributes that are associated with a namespace URI must have a prefix (default namespaces do not apply to attributes). This conforms to section 5.2 of the "W3C Namespaces in XML" recommendation. If an attribute references a namespace URI, but does not specify a prefix, the writer generates a prefix for the attribute.

When writing an empty element, an additional space is added between the tag name and the closing tag: for example, **<item />**. This provides compatibility with older browsers.

When a **String** is used as a method parameter, a reference to **Nothing** and **String.Empty** is equivalent. **String.Empty** follows the W3C rules.

You can also write strongly typed data by using the **XmlConvert** class which works just like the standard **Convert** class. For example, the following line of code converts **String** data to **Double** data:

```
Dim total As Double = XmlConvert.ToDouble(reader.ReadInnerXml())
```

There is a ton of XML specific information that would be terrific to cover but much of it belongs in a book dedicated to the subject. However, there is one more I/O facility we need to look at in this book before we can close the long chapter: serialization.

Serialization with XML

Serialization support in .NET is extensive. The base implementations and interfaces are derived from **System.Runtime.Serialization**. The **System.Runtime.Serialization.Formatters** namespace provides enumeration support and the base classes for the serialization formatting. Then we have access to two namespaces that provide the actual implementation and formatting. One class provides binary formatting (**Formatters.Binary**) and the other, which we are going to use, provides text formatting in SOAP format (**Formatters.Soap**).

The **Soap** class is very useful for serializing across and through network boundaries, because the data, XML, is encapsulated in a SOAP envelope. The framework also provides a namespace specializing in pure XML serialization streams, essentially serializing into and out of XML documents.

Serialization is very useful for persisting data stored in various data structures. It's also very lightweight and efficient to use, especially for applications running on Web servers, database servers, and various facilities that you may need to scale. One scenario where serialization comes in handy is loading data into an application at start up. The data can be easily piped into an object that is created on the fly at runtime. The Indexworks application introduced at the beginning of the chapter is one such application.

The outmoded way of loading data into the application at runtime would have you create an array or some other data structure, initialize the structure, open a flat text file, read the data into the array, and then position the array for access by the application's various components. The modern approach instead lets you suck in (serialize) the XML file into the application's processing space, creating and initializing the object that holds the data all at the same time.

The lattter sophisticated approach cuts out the step of having to read in the data and add it to the elements of an array or the nodes of a linked list one element or node at a time. It's like having the entire object and all its data sitting on the disk ready for loading at a moment's notice (in fact it's exactly that).

tablet, on the other side of the galaxy, on a console, or in another reality, without affecting the business logic in any way whatsoever.

This disconnection is exactly how ASP.NET applications and rich−client applications will work as we move into mainstream .NET development. All it takes to move a desktop application to the Internet is to drop the form−based UI running on the client, and replace it with a Web form running on Internet Explorer.

The separation of the two application "domains" allows an effective development team to keep its best UI people working on the front end while the logic and objects in the operational side of the application can be worked on by the best class and object designers and code construction workers. As long as the "back−end" developers understand that they are creating classes and objects that clients' code will "hook" into, you can achieve a highly cohesive and productive software development workforce that lets developers that have an artistic flair do a lot more that just screen painting.

There is no longer an excuse not to delegate properly with Visual Basic .NET and the power of the .NET Framework. There is no excuse not to extend specialized and generic collections of classes with inheritance, and there is no excuse not to delegate and use the power of polymorphism with native interfaces and delegates.

A Windows forms program can be a stand−alone executable or exist as the client portion of a multitiered system. There are various ways of connecting to the back−end logic, and the most cutting−edge method is via Web services technology over HTTP. The server typically can be connected to a database, a mail server, or any other collection of objects you care to call "server." The Windows forms technology is such that your new featherweight classes can act as the UI to a powerful, data−enabled system that leverages the rich UI of a client application with the advanced processing of an application server. To encapsulate this is a single utterance: "The Web is dead, long live the Web."

Windows Forms

The Windows Forms technology is the new UI solution for the .NET Framework. All UI elements, such as forms and visual input, output, and presentation components, extend a hierarchy of classes found in the **System.Windows.Forms** namespace. You can use the forms classes and controls classes as is, or derive from them to create your own UI and visual controls and components. The Windows forms are an ideal OOP solution for creating rich UIs for local workstation clients, or as thin UIs developed for multitier distributed solutions.

Windows−based UIs are typically cast in the following three styles:

- **Single document interface (SDI)** This is the UI that only opens a single document, such as Notepad or WordPad or Outlook, which opens e−mails. You first have to close the current document before you can open a new one. You can use the SDI application for simple document editors, various utilities, and applications that do not need to work with multiple open forms.
- **Multiple document interface (MDI)** This UI contains numerous forms that encapsulate documents, database input fields, grids, drawing areas, and various layouts and components. You can open new forms in the UI as you need them. You do not need to close forms before opening new ones. Forms inside the main form are enumerated into the Window menu for easy management and access. A good example of the MDI application is Microsoft Word.
- **Explorer−style interface** This UI is an SDI application that is split into two panes inside a single parent form. The left pane provides access to a tree of items, such as the so−called "cool" bar or some other type of collection. The right pane provides the details of the node selected from the tree. A good

example of this type of interface is Microsoft Outlook or Windows Explorer.

A form is your application's little claim of screen space that you will use to communicate with users. Your form will occupy either a portion of the screen or all of it. Forms are typically rectangular in shape and can be made to shrink or grow to any sizefrom the size of a dime or less, to the size of the screen when the form is fully maximized. Forms can be solid, opaque, or invisible. With advanced support of the Windows XP and .NET Server operating systems, forms can also now be any shape.

You use the form to present information to the user and to accept input from the user. This is achieved by placing familiar objects on the form, with which the user interacts to send and receive information to and from the application.

UI developers arrange the controls on the form in an aesthetic and productive manner by exposing the various properties of the input/output controls placed on the form. The properties of the controls define their behavior and affect how the user interacts with the application.

As the interface developer, you will also spend a substantial amount of your time behind the form's UI, implementing its code. Controls do not magically hook into existing functionality you have written; you still have to "wire up" the UI to the back end and business logic and hook up the events to the event handlers and event listeners. This wiring involves capturing events generated by the controls on the form, such as text being entered, mouse clicks, button clicks, scrolling through lists, collapsing and expanding trees, and so on. The events communicate with interested objects that monitor the form for services they need to perform, such as sending data to a database and retrieving data from the database.

A Form Is an Object

As previously stated many times, a class is a template or a blueprint for an object. The .NET Form class is such a blueprint for the form object that you instantiate. **Form** classes can also be extended by you. This means you can easily inherit from existing framework or custom forms to add functionality or modify existing behavior. In other words, when you add a form to your project, you can choose to inherit from the standard **Form** class or from a custom base **Form** class you may have already developed. The form hierarchy of classes is a perfect example of how inheritance is used as the foundation for all specializations of classes in an OO framework. **Form** objects are also controls, because the standard form provided by the framework ultimately inherits from its parent **Control** class.

Chapter 9 presented a concise introduction to the **Form** class in the discussion of inheritance and aggregation. We saw then how a form can be created entirely in the Code Editor. But Visual Studio makes it far easier to use the Windows Forms Designer to create and modify forms. Later in this section, we will discuss the steps to take to kick off a UI project with the creation of your main form, and any collateral forms used by your application.

The System.Windows.Forms Namespace

The **System.Windows.Forms** namespace contains the collection of classes used for creating Windows−based UI applications. The classes in this namespace can be grouped as follows:

- Control
- UserControl
- Form
- Controls

You can also set the following properties while in this method, or from the Properties window:

```
Me.Name = "MainForm"
Me.Text = "Indexworks"
Me.WindowState = FormWindowState.Maximized
```

At this point, it makes life easier if you rename the actual source code file to something more meaningful than Form1.vb. I changed the name to MForm (for main form), which tells me that the form contains the source for the parent form. I also renamed the class file for the form to **MainForm**, which is more useful than **Class1**. The **Text** property shown in the preceding code is the caption at the top of the parent form, which you reserve for the name of the application.

It is easier to work with MDI child windows when the parent form is maximized. You will also notice that the edges of the MDI parent form will be the same as the system color, which you set in the Windows System control panel. This property is not affected by the back color set using the **Control.BackColor** property. At this juncture, if the form is too small in the designer, you can make it bigger by dragging with the mouse, or you can set the height and width with the following code in the **InitializeComponent** method:

```
Me.ClientSize = New System.Drawing.Size(600, 400)
```

Once this is done, you can add the first menu resources to the parent form as follows:

1. Switch to visual mode and click the form so that the Toolbox becomes active.
2. Drag or double−click a **MainMenu** component from the Toolbox palette to the form. The first thing you'll notice is that the menu hides as soon as it hits the form. You can get it back from the drop−down list at the top of the Properties window.
3. Click the menu component's top−level menu item and set the property to **&File**. You can also create sub−submenu items like **&New**, **&Close**, and **&Exit** in the same manner. And you can also create top−level menu items called **&Window** and **&Help**, although a Help menu may be a long way off at this stage from being implemented (nothing wrong with starting the Help system at the beginning of the development; after all, if you followed my advice in the past chapters, most of the applicationthe code behindhas already been built). The File menu items are where you write the code to create, and open windows, and the Close menu will be used to close down the application. The Window menu will be used to keep track of open MDI child windows that are enumerated. The menus and the form built at this point are illustrated in Figure 16−4.

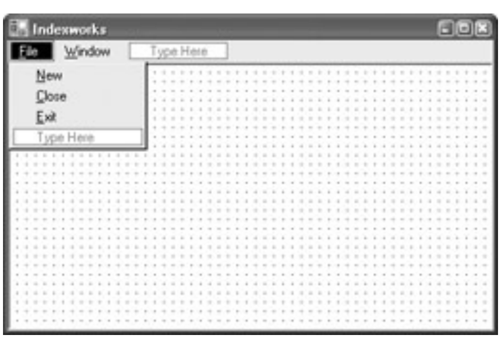


Figure 16−4: The new MDI parent form and initial menus

Tip Rename the menus from the default identifiers, like MenuItem2, that Visual Studio assigns. This will make it easier for you to find and set properties in the correct menu item later when the number of menus listed in the Properties window grows. You also need more intuitive menu names for the event wiring that comes later when you connect the click on a menu item to an

```
   ' Get the active child form.
   Dim activeChild As Form = Me.ActiveMDIChild

   ' If there is an active child form, find the active control, which
   ' in this example should be a RichTextBox.
   If (Not activeChild Is Nothing) Then
      Try
         Dim textBox As TextBox = _
         Ctype(activeChild.ActiveControl, RichTextBox)
         If (Not textBox Is Nothing) Then
            ' Put selected text on Clipboard.
            Clipboard.SetDataObject(textBox.SelectedText)
         End If
      Catch
         MessageBox.Show("Please select TextBox.")
      End Try
   End If
End Sub
```

Arranging the Forms

If you provide the user with the ability to open more than one form, you'll want to provide the ability to arrange the forms automatically. The built−in options you have for arranging all the forms as a collection are Tile, Cascade, and Arrange.

You can provide the arranging facility by reference any one of the **MDILayout** enumeration values that cause the child forms to arrange as you specify. The enumeration values let you arrange the child forms as cascading, as horizontally or vertically tiled, or as child form icons that are fanned out along the lower portion of the MDI form in a minimized state.

You can also use constructs such as the event handlers called by a menu item's **Click** event. This lets you create a menu item, such as Cascade Windows, that provides the effect of cascading child MDI child windows.

To arrange child forms, create a method to set the **MDILayout** enumeration for the parent. The following code demonstrates referencing the **Cascade** constant of the **MDILayout** enumeration for the child windows of the MDI parent form. You will typically use the enumeration in your code as follows:

```
Protected Sub CascadeWindows_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs)
  MainForm.LayoutMDI(System.Windows.Forms.MDILayout.Cascade)
End Sub
```

That's about as far as I need to take you with MDI applications. The rest of the chapter explores the various UI elements you can use for building out your MDI application.

Delegating Application Startup and Shutdown

The life of your application typically begins and ends with the parent or main form. When you close the main form, you terminate the life of the application. However, your main form does not have to become the controlling object in the life of your application. You can relocate the entry point and delegate the application's start up and shut down code to other objects. This can be achieved by moving the startup logic into a separate object that only you know exists somewhere in the vast expanse of memory called the heap.

Changing Borders

We can determine the look and feel of our forms by setting various border properties. This is done by changing the **FormBorderStyle** property. The property also lets you control the resizing behavior of the form, how the caption bar is displayed, and what buttons might appear on the form. Table 16−2 lists the property settings for changing borders.

All the border styles listed in Table 16−2, with the exception of **None**, provide the Close box on the right side of the title bar. You can set the border style of the form interactively with the **FormBorderStyle** property.

The border style of the form is set using the **FormBorderStyle** enumeration, so you can also easily set the border style programmatically by setting the **FormBorderStyle** property to one of the values of the enumeration. The following code provides an example:

```
SearchDlgBx.FormBorderStyle = FormBorderStyle.FixedDialog
```

This code sets the form to a border style that lets the form have Minimize and Maximize buttons. You can also specify whether you would like either or both of these to be functional; however, the Minimize and Maximize buttons are enabled by default, and their functionality is manipulated through property settings as well.

Table 16−2: Border Styles for Forms

Setting	Purpose
None	The form contains no border or border−related elements. It is used for startup forms.
Fixed 3D Not resizable	For 3−D border effects. You can include control−menu box, title bar, Maximize and Minimize buttons on the title bar. Provides a raised border relative to the body of the form.
Fixed Dialog Not resizable	For dialog boxes. It can include control−menu box, title bar, Maximize and Minimize buttons on the title bar. Provides a recessed border relative to the body of the form.
Fixed Single Not resizable	Can include control−menu box, title bar, Maximize button, and Minimize button. Resizable only using Maximize and Minimize buttons. Creates a single line border.
Fixed Tool	For tool windows. It displays a nonsizable window with a Close button and title bar text in a reduced font size. This form does not appear in the Windows taskbar.
Sizable (Default)	Use as main window and child windows. The form is resizable and can include control−menu box, title bar, Maximize button, and Minimize button. It can be resized using the control−menu box, Maximize button, and Minimize button on the title bar, or by using the mouse pointer on any edge.
Sizable Tool	A sizable tool window displays sizable with a Close button and title bar text in a reduced font size. The form does not appear in the Windows taskbar.

To disable the Minimize and Maximize buttons set the **MinimizeBox** or **MaximizeBox** properties to **False** (properties such as this are inherited, and it is thus easier to set them interactively in the Properties editor).

click the pixels you want to designate as the cursor's hot spot. The **Hotspot** property in the Properties window displays the new coordinates.

Tip Tooltips can be made to appear when you hover your cursor over a toolbar button. These tips can help you identify the function of each button.

Mouse and Keyboard Events

Mice can do a lot more than nibble cheese. In a Windows UI, they can let you know when one of their buttons has been clicked or released, or when the mouse pointer leaves or enters some part of the application. This information is provided in the form of **MouseDown**, **MouseUp**, **MouseMove**, **MouseEnter**, **MouseLeave**, and **MouseHover** events.

KeyPress events also bubble up from the OS and are made available to you in **KeyPress**, **KeyDown**, and **KeyUp** events. Mouse event handlers receive an argument of type **EventArgs** containing data related to their events; however, key−press event handlers receive an argument of type **KeyEventArgs** (a derivative of **EventArgs**) containing data specific to the keypress or key release events.

When wiring up mouse events, you can also change the mouse cursor. You typically marry the ability to change the cursor to the **MouseEnter** and **MouseLeave** events. These are used to provide feedback to the user that something is happening, not happening, or that certain areas are offlimits or welcome to the explorative nature of your cursor. The event is exposed in the following code example:

```
Public Event MouseDown As MouseEventHandler
```

Table 16−3 lists the **MouseEventArgs** properties to provide information specific to the mouse event.

Mouse events occur in the following order:

1. MouseEnter
2. MouseMove
3. MouseHover/MouseDown/MouseWheel
4. MouseUp
5. MouseLeave

Table 16−3: *MouseEventArgs* Properties

Property	Purpose
Button	Tells you which mouse button was pressed
Clicks	Tells you how many times the mouse button was pressed and released
Delta	Retrieves a signed count of the number of *detents* the mouse wheel has rotated. A detent is one notch of the mouse wheel.
X	Retrieves the X coordinate of a mouse click
Y	Retrieves the Y coordinate of a mouse click

Calendar

The **MonthCalendar** provides an intuitive graphical interface in the form of a calendar to allow you to view and set date information. The control displays a calendar that contains the numbered days of the month, arranged in columns underneath the days of the week, with the selected range of dates highlighted. The control is illustrated in Figure 16−13.

Figure 16−13: The MonthCalendar control

The user can select a different month by clicking the arrow buttons that appear on either side of the month caption. This control lets the user select more than one date, whereas the **DateTimeControl** does not.

You might consider using a **DateTimePicker** control instead of a **MonthCalendar** if you need custom and possibly programmatic date formatting or need to enforce a single selection for input.

A Palette

The **ColorDialog** component is a preconfigured dialog box containing a palette that lets the user select a color from the palette and to add custom colors to that palette. This is a common dialog box that you see in all other Windows applications that offer color selection, including Visual Studio. It makes sense to use this palette instead of configuring a new palette.

Like the other common dialog boxes, this control has certain read/write properties that will be set to default values. You can, however, change these values in the dialog box's constructor.

List Boxes

The **ListBox** control is an old favorite in Windows applications. It displays a list of items to your users and allows them to select one or more items. This control has an embedded vertical scroll box that is displayed if the total number of items exceeds the number that can be displayed. The control can also show multiple columns when you set its **MultiColumn** property to **True**. If you set the **ScrollAlwaysVisible** to **True**, the scroll bars appear regardless of the number of items or columns. You can also code against the **SelectionMode** property to determine the number of list items that can be selected at a time.

CheckedListBox

The **CheckedListBox** control extends the **ListBox** control with the ability to check off items in the lists. The checked list boxes can only have one item, but a selected item is not the same thing as a checked item. These controls can also be data bound, like list boxes, by programming against the **DataSource** property. They can also obtain their items from a collection, using the **Items** property.

ListView

The **ListView** control displays a list of items with the option of including an icon with each item. The typical use of the **ListView** control is to create the details facility in a Windows Explorerstyle application. There are four modes to use with the basic version of this control: LargeIcon, SmallIcon, List, and Details. LargeIcon mode displays large icons next to the item text; the items appear in multiple columns if the control is large enough. SmallIcon mode is just a small−icon version of LargeIcon mode.

List mode displays the small icons, and the list is presented as a single column. Details mode shows the items in multiple columns with details represented in the columns. You can add columns to this control in your code. You also have the option of setting the **View** property in this control. The view modes provide the ability to display images from image lists. See the SDK for more specifics.

Trackbars (Sliders)

TrackBar controls, often known as "slider" controls, are used mostly for adjusting a numeric value. The **TrackBar** control has two parts: the slider, or thumb, and the notches. The slider is the part that can be adjusted by sliding from side to side or up and down. Its position on the control provides the facility to return the **Value** property. The notches indicate a range of values placed at evenly spaced positions on the scale.

Toolbars

The **ToolBar** control is used as a staging area for displaying a row of drop−down menus and bitmapped buttons. Toolbar buttons may be mapped to menu item commands and can be configured to appear and behave as push buttons, drop−down menus, or separators. Typically, a toolbar provides quick access to the application's most frequently used facilities.

A **ToolBar** control may be "docked" along the top of its parent window, which is its usual place. It may also be docked to any side of the window, or it may float. You can also change the size of the **ToolBar** and drag it. The toolbar can display tooltips. To display **ToolTip**s, the **ShowToolTips** property of the control must be set to **True** (see "**ToolTip**" later in this chapter).

TreeView

The **TreeView** control displays a hierarchy of tree nodes exactly like the hierarchy of classes in the Object Browser. Each node can contain child nodes and parent nodes, and child nodes can themselves be parent nodes. The tree can also be expanded or collapsed.

The **TreeView** control also provides the ability to display check boxes next to the nodes. This can be done by programming against the tree view's **CheckBoxes** property. Selecting or clearing nodes is achieved by setting the node's **Checked** property to **True** or **False**.

Presentation and Informational Controls

Some controls and components are designed to present information to users. These include the controls listed in Table 16−6.

"It doesn't work!"

Thinking in Debug Terms

Before you can debug a .NET Framework application, the compiler and the run−time environment must be configured for the debug "state of mind." Visual Studio does this for you automatically when you place your application in Debug mode, as described shortly. This configuration is essential to enable a debugger to attach to the application for the purpose of producing symbols and line maps for your source code and the Microsoft Intermediate Language (MSIL) that presents it to the CLR.

Released software, which is debugged for release candidate builds, can then be profiled to boost performance. The job of the profiler, a software tool, is to evaluate and describe the lines of source code that generate the most frequently executed code, and then estimate how much time it takes to execute them.

In addition to Visual Studio's debugging utilities, you can examine and improve the performance of .NET Framework applications using the following resources:

- **Classes in the Systems.Diagnostics namespace** This chapter investigates the **Debug** and **Trace** classes in this namespace in some depth.
- **Runtime Debugger (Cordbg.exe)** This is Microsoft's standard .NET command−line debugger, which is not covered in this book.
- **Microsoft Common Language Runtime Debugger (DbgCLR.exe)** This debugger ships with the .NET SDK and is not covered in this book.

You can use the **System.Diagnostics** classes for tracing execution flow, and you can use the **Process**, **EventLog**, and **PerformanceCounter** classes for profiling code. You can also use the Cordbg.exe command−line debugger to debug managed code from the command−line interpreter. If you prefer not to labor on the command line, the DbgCLR.exe can be accessed in a familiar Windows interface. Both compilers are used for debugging managed code. The latter is located in the %\FrameworkSDK\GuiDebug folder, while the former can be found in the Microsoft Visual Studio .NET\FrameworkSDK\Bin folder.

The System.Diagnostics Namespace

To get you on the road to proving that your code works, the .NET Framework provides a namespace jam−packed with classes and various components specifically designed to allow you to interact with system processes, event logs, performance counters, and other run−time elements. This namespace also includes a collection of services used with thread management (see Chapter 16) and a special class**Debug**specifically used to help debug your code on a line−by−line basis. The **Debug** class is further discussed later in this section.

Tables 17−1 and 17−2 list the resources in this namespace and briefly describe the services they provide.

Table 17−1: Base and Final Classes in the *System.Diagnostics* Namespace

Class [*]	Description
BooleanSwitch	A **Boolean** construct that you can use for conditional elements in your code. It provides conditions logic for

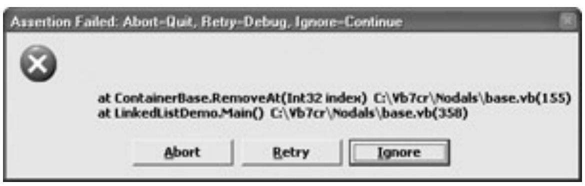
	control debugging and tracing output.
ConditionalAttribute	An attribute that indicates to compilers that a method is callable only if a specified preprocessing identifier is applied to it. This attribute is thus especially useful for ensuring the compiler keeps certain debug information in the assembly during run time, because debug information is typically stripped out in the release build.
CounterCreationData	Used to define and create custom counter objects. With this class, you can specify the counter type, name, and help string for a custom counter.
CounterCreationDataCollection	Used to create strongly typed collections of **CounterCreationData** objects
CounterSampleCalculator	Contains a single static method for computing raw performance counter data
Debug	The main debug class that provides a set of methods and properties that you will use as an aid in debugging code
DebuggableAttribute	An attribute that modifies code generation for run−time just−in−time (JIT) debugging. This attribute can be used to specify how the CLR gathers debug information at run time.
Debugger	Allows you to communicate directly with the debugger. For example, it contains a method called **Launch** that fires up the debugger from within your code.
DebuggerHiddenAttribute	Specifies the **DebuggerHiddenAttribute**
DebuggerStepThroughAttribute	Specifies the **DebuggerStepThroughAttribute**
DefaultTraceListener	The main class that provides the default output methods and behavior for tracing (see "Tracing and the Trace Class" later in this chapter)
EntryWrittenEventArgs	The event data target for the **EntryWritten** event
EventLog	Provides the interaction with Windows event logs
EventLogEntry	Encapsulates a single record in the event log
EventLogEntryCollection	Defines size and enumerators for a collection of **EventLogEntry** instances
EventLogInstaller	Used for installing and configuring an event log that your application reads from or writes to when running. This class can be used by an installation utility (for example, InstallUtil.exe) when installing an event log
EventLogPermission	Allows control of code access permissions for event logging
EventLogPermissionAttribute	Contains the attribute for allowing declarative permission checks for event logging
EventLogPermissionEntry	Defines the smallest unit of a code access security permission that is set for an **EventLog**
EventLogPermissionEntryCollection	Contains a strongly typed collection of **EventLogPermissionEntry** objects
EventLogTraceListener	

are included in the **System.Diagnostics** namespace). **Debug** class methods are not included in a release version of your program, so they do not increase the size or reduce the speed of your release code.

In the preceding methods, the first argument, which is mandatory, represents the condition that you want to check. If you call **Assert** with only one argument, as described in the preceding list, the method will check the condition and, if it turns out to be **False**, transmit the contents of the call stack to the Assert message box. The following example demonstrates calling **Assert** and passing one argument to the single−parameter version:

```
Debug.Assert(index < 0)
```

In this case the **index** value was greater than 0 and the message box shown in the illustration is displayed.



You can see the **Assert** at work in the following code snippet at a location that might make sense as an exception throw point:

```
Public Sub RemoveAt(ByVal index As Integer) Implements IList.RemoveAt
  Try
    Debug.Assert(index <= 0)
      If (index < 0 Or index >= Count) Then
       Throw New ArgumentOutOfRangeException()
     End If
'...
End Sub
```


Tip If you prefer a less obnoxious failure notification than the **Assert** message box there are a few things you can do. The **DefaultTraceListener** (see the "Tracing and the Trace Class" section coming up) controls the output for the **Assert** method so you can turn off the message box (see the "Setting Assertions in Configuration Files" towards the end of this section. The output can be directed to the Debug Output window, other listeners, log files, and so on. You can also control the listeners directly in your code (see the section "Tracing and the Trace Class" coming up).

As you can see from the following examples, both second and third parameters take **String** arguments. Calling **Assert** with two or three arguments forces **Assert** to check the condition and, if the result is **False**, output one or two **String**s to the Output window. The following examples demonstrate calling **Assert** and passing in two or three arguments:

```
Debug.Assert(index < 0, "Index points to nothing at the tail.")
Debug.Assert(index > count  1), "Index points to nothing at the _
  head.", Format(size, "G"))
```

You will use the **Debug.Assert** method to test conditions that should hold true if your code is correct. In the following example, I used **Assert** to debug the Singleton pattern implementation:

```
Shared Function GetInstance() As Singleton
  Singleton = New Singleton()
  Singleton.Number += 1
  Debug.Assert(Singleton.Number > 1)
  If (Singleton.Number > 1) Then
    'I have no exception to use here yet
```

- **Me window** This window displays the data members of the current object containing the method the code is currently executing.
- **Memory windows** These windows (there are four of them) can be used to view large buffers, strings, and other data that does not display well in the Watch or Variables window.
- **Modules window** This window provides information about the DLL files and EXE used by the current application.
- **Registers window** The Registers window displays the contents of a register. It is useful to keep the Registers window open as you step through your code because it lets you see register values change as your code executes.
- **Running Documents window** This window provides information related to any script files used by the current application.
- **Threads window** This window lets you access the threads that are currently running in your application.
- **Watch window** The Watch window is used to evaluate variables and expressions and keep the results. The Watch window can also be used to edit the value of a variable or register.
- **QuickWatch dialog box** The QuickWatch dialog box can be used to quickly evaluate a variable or expression. It is simpler to use than the Watch window.

Note Edit and Continue is a feature of Visual Studio that lets you change your source code while your program is in Break mode and then apply those changes without having to end the debug session and rebuild your program. This feature was available to Visual Studio 6 and was made available to Visual Basic 6 programmers. Unfortunately, it is a feature only available to C++ programmers as of the first release of Visual Studio .NET, and we pray it will return to the Visual Basic language side of the house as soon as possible.

Getting Started

Your first debugging activity will likely be to set breakpoints in your code and step through your application. The following is the order of the steps you will take through your code:

1. Start execution.
2. Break execution (halt execution).
3. Continue (resume execution).
4. Stop execution.
5. Step through the application.
6. Run to a specified location.
7. Set the execution point.

To set a breakpoint in your code, place your cursor at a valid line for a breakpoint and then double−click in the left margin of the source code editor. A breakpoint, represented by a large round blob, is inserted, as illusrated here.

Alternatively, if you don't have the source code file open in front of you, you can set a breakpoint in the New Breakpoint dialog box. As illustrated in Figure 17−2, this dialog box can be opened by selecting the Debug menu and choosing New Breakpoint.