C# PROGRAMMING AND .NET

PART – A

UNIT – 1 6 Hours

The Philosophy of .NET: Understanding the Previous State of Affairs, The.NET Solution, The Building Block of the .NET Platform (CLR,CTS, and CLS), The Role of the .NET Base Class Libraries, What C# Brings to the Table, An Overview of .NET Binaries (aka Assemblies), the Role of the Common Intermediate Language, The Role of .NET Type Metadata, The Role of the assembly Manifast, Compiling CIL to Platform – Specific Instructions, Understanding the Common Type System, Intrinsic CTS Data Types, Understanding the Common Languages Specification, Understanding the Common Language Runtime A tour of the .NET Namespaces, Increasing Your Namespace Nomenclature, Deploying the .NET Runtime.

UNIT – 2 6 Hours

Building C# Applications: The Role of the Command Line Complier (csc.exe), Building C# Application using csc.exe Working with csc.exe Response Files, Generating Bug Reports, Remaining g C# Complier Options, The Command Line Debugger (cordbg.exe) Using the, Visual studio .NET IDE, Other Key Aspects of the VS.NET IDE, C# "Preprocessor:" Directives, an Interesting Aside: The System. Environment Class.

UNIT – 3 8 Hours

C# Language Fundamentals: The Anatomy of Basic C# Class, Creating objects: Constructor Basics, The Composition of a C# Application, Default assignment and Variable Scope, The C# Member Initialisation Syntax, Basic Input and Output with the Console Class, Understanding Value Types and Reference Types, The Master Node: System, Object, The System Data Types (and C# Aliases), Converting Between Value Types and Reference Types: Boxing and Unboxing, Defining Program Constants, C# Iteration Constructs, C# Controls Flow Constructs, The Complete Set of C# Operators, Defining Custom Class Methods, Understating Static Methods, Methods Parameter Modifies, Array Manipulation in C#, String Manipulation in C#, C# Enumerations, Defining Structures in C#, Defining Custom Namespaces.

UNIT – 4 6 Hours

Object- Oriented Programming with C#: Forms Defining of the C# Class, Definition the "Default Public Interface" of a Type, Recapping the Pillars of OOP, The First Pillars: C#'s Encapsulation Services, Pseudo- Encapsulation: Creating Read-Only Fields, The Second Pillar: C#'s Inheritance Supports, keeping Family Secrets: The " Protected" Keyword, Nested Type Definitions,The Third Pillar: C #'s Polymorphic Support, Casting Between .

PART – B

UNIT – 5 6 Hours

Exceptions and Object Lifetime: Ode to Errors, Bugs, and Exceptions, The Role of .NET Exception Handing, the System. Exception Base Class, Throwing a Generic Exception, Catching Exception, CLR System – Level Exception(System. System Exception), Custom Application-Level Exception(System. System Exception), Handling Multiple Exception, The Family Block, the Last Chance Exception Dynamically Identifying Application – and System Level Exception Debugging System Exception Using VS. NET, Understanding Object Lifetime, the CIT of "new', The Basics of Garbage Collection,, Finalization a Type, The Finalization Process, Building an Ad Hoc Destruction Method, Garbage Collection Optimizations, The System. GC Type.

UNIT – 6 6 Hours

Interfaces and Collections: Defining Interfaces Using C# Invoking Interface Members at the object Level, Exercising the Shapes Hierarchy, Understanding Explicit Interface Implementation, Interfaces As Polymorphic Agents, Building Interface Hierarchies, Implementing, Implementation, Interfaces Using VS .NET, understanding the IConvertible Interface, Building a Custom Enumerator (IEnumerable and Enumerator), Building Cloneable
objects (ICloneable), Building Comparable Objects (I Comparable), Exploring the system. Collections Namespace, Building a Custom Container (Retrofitting the Cars Type).

UNIT – 7 8 Hours

Callback Interfaces, Delegates, and Events, Advanced Techniques: Understanding Callback Interfaces, Understanding the .NET Delegate Type, Members of System. Multicast Delegate, The Simplest Possible Delegate Example, , Building More a Elaborate Delegate Example, Understanding Asynchronous Delegates, Understanding (and Using)Events.

The Advances Keywords of C#, A Catalog of C# Keywords Building a Custom Indexer, A Variation of the Cars Indexer Internal Representation of Type Indexer . Using C# Indexer from VB .NET. Overloading operators, The Internal Representation of Overloading Operators, interacting with Overload Operator from Overloaded-Operator- Challenged Languages, Creating Custom Conversion Routines, Defining Implicit Conversion Routines, The Internal Representations of Customs Conversion Routines

UNIT – 8 6 Hours

Understanding .NET Assembles: Problems with Classic COM Binaries, An Overview of .NET Assembly, Building a Simple File Test Assembly, A C#. Client Application, A Visual Basic .NET Client Application, Cross Language Inheritance, Exploring the CarLibrary's, Manifest, Exploring the CarLibrary's Types, Building the Multifile Assembly, Using Assembly, Understanding
Private Assemblies, Probing for Private Assemblies (The Basics), Private Assemblies XML Configurations Files, Probing for Private Assemblies (The Details), Understanding Shared Assembly, Understanding Shared Names, Building a

Shared Assembly, Understanding Delay Signing, Installing/Removing Shared Assembly, Using a Shared Assembly

Text Books:

1. Andrew Troelsen: Pro C# with .NET 3.0, 4th Edition, Wiley India, 2009. Chapters: 1 to 11 (up to pp. 369)
2. E. Balagurusamy: Programming in C#, 2nd Edition, Tata McGraw Hill, 2004. (Programming Examples 3.7, 3.10, 5.5, 6.1, 7.2, 7.4, 7.5, 7.6, 8.1, 8.2, 8.3, 8.5, 8.7, 8.8, 9.1, 9.2, 9.3, 9.4, 10.2, 10.4, 11.2, 11.4, 12.1, 12.4, 12.5, 12.6, 13.1, 13.2, 13.3, 13.6, 14.1, 14.2, 14.4, 15.2, 15.3, 16.1, 16.2, 16.3, 18.3, 18.5.18.6)

Reference Books:

1. Tom Archer: Inside C#, WP Publishers, 2001.
2. Herbert Schildt: C# The Complete Reference, Tata McGraw Hill, 2004.

Table of Content Page no

Introducing C# and the .NET Platform
UNIT-1

UNIT -1 The philosophy of .NET

1.1 Understanding the Previous State of Affairs
1.2 The .NET Solution,
1.3 The Building Block of the .NET Platform (CLR,CTS, and CLS)
1.4 The Role of the .NET Base Class Librarie
1.5 What C# Brings to the Table
1.6 An Overview of .NET Binaries
1.7 The Role of the Common Intermediate Language
1.8 The Role of .NET Type Metadata
1.9 The Role of the Assembly Manifest
1.10 Compiling CIL to Platform –Specific Instructions
1.11 Understanding the Common Type System
1.12 Intrinsic CTS Data Types,
1.13 Understanding the Common Languages Specification,
1.14 Understanding the Common Language Runtime
1.15 A tour of the .NET Namespaces,

1.1 Understanding the Previous State of Affairs

Before examining the specifics of the .NET universe, it's helpful to consider some of the issues that motivated the genesis of Microsoft's current platform. To get in the proper mind-set, let's begin this chapter with a brief and painless history lesson to remember our roots and understand the limitations of the previous state of affairs (after all, admitting you have a problem is the first step toward finding a solution). After completing this quick tour of life as we knew it, we turn our attention to the numerous benefits provided by C# and the .NET platform.

Life As a C/Win32 API Programmer

Traditionally speaking, developing software for the Windows family of operating systems involved using the C programming language in conjunction with the Windows application programming interface (API). While it is true that numerous applications have been successfully created using this time-honored approach, few of us would disagree that building applications using the raw API is a complex undertaking.

The first obvious problem is that C is a very terse language. C developers are forced to contend with manual memory management, ugly pointer arithmetic, and ugly syntactical constructs. Fur thermore, given that C is a structured language, it lacks the benefits provided by the object-oriented approach (can anyone say *spaghetti code*?) When you combine the thousands of global functions and data types defined by the Win32 API to an already formidable language, it is little wonder that there are so many buggy applications floating around today.

Life As a C++/MFC Programmer

One vast improvement over raw C/API development is the use of the C++ programming language. In many ways, C++ can be thought of as an object-oriented *layer* on top of C. Thus, even though C++ programmers benefit from the famed "pillars of OOP" (encapsulation, inheritance, and polymorphism), they are still at the mercy of the painful aspects of the C language (e.g., manual memory management, ugly pointer arithmetic, and ugly syntactical constructs). Despite its complexity, many C++ frameworks exist today. For example, the Microsoft Foundation Classes (MFC) provides the developer with a set of C++ classes that facilitate the construction of Win32 applications. The main role of MFC is to wrap a "sane subset" of the raw Win32 API behind a number of classes, magic macros, and numerous code-generation tools (aka *wizards*). Regardless of the helpful assistance offered by the MFC framework (as well as many other C++-based windowing toolkits), the fact of the matter is that C++ programming remains a difficult and error-prone experience, given its historical roots in C.

Life As a Visual Basic 6.0 Programmer

Due to a heartfelt desire to enjoy a simpler lifestyle, many programmers have shifted away from the world of C(++)-based frameworks to kinder, gentler languages such as Visual Basic 6.0 (VB6). VB6 is popular due to its ability to build complex user interfaces, code libraries (e.g., COM servers), and data access logic with minimal fuss and bother. Even more than MFC, VB6 hides the complexities of the raw Win32 API

from view using a number of integrated code wizards, intrinsic data types, classes, and VB-specific functions.
The major downfall of VB6 (which has been rectified given the advent of Visual Basic .NET) is that it is not a fully object-oriented language; rather, it is "object aware." For example, VB6 does not allow the programmer to establish "is-a" relationships between types (i.e., no classical inheritance) and has no intrinsic support for parameterized class construction. Moreover, VB6 doesn't provide the ability to build multithreaded applications unless you are willing to drop down to low-level Win32 API calls (which is complex at best and dangerous at worst).

Life As a Java/J2EE Programmer
Enter Java. The Java programming language is (almost) completely object oriented and has its syntactic roots in C++. As many of you are aware, Java's strengths are far greater than its support for platform independence. Java (as a language) cleans up many unsavory syntactical aspects of C++. Java (as a platform) provides programmers with a large number of predefined "packages" that contain various type definitions. Using these types, Java programmers are able to build "100% Pure Java" applications complete with database connectivity, messaging support, web-enabled front ends, and a rich user interface. Although Java is a very elegant language, one potential problem is that using Java typically means that you must use Java front-to-back during the development cycle. In effect, Java offers little hope of language integration, as this goes against the grain of Java's primary goal (a single programming language for every need). In reality, however, there are millions of lines of existing code out there in the world that would ideally like to commingle with newer Java code. Sadly, Java makes this task problematic. Pure Java is simply not appropriate for many graphically or numerically intensive applications hese cases, you may find Java's execution speed leaves something to be desired). A better approach for such programs would be to use a lower-level language (such as C++) where appropriate. Alas, while Java does provide a limited ability to access non-Java APIs, there is little support for true cross-language integration.

Life As a COM Programmer
The Component Object Model (COM) was Microsoft's previous application development framework. COM is an architecture that says in effect, "If you build your classes in accordance with the rules of COM, you end up with a block of *reusable binary code*."
The beauty of a binary COM server is that it can be accessed in a language-independent manner. Thus, C++ programmers can build COM classes that can be used by VB6. Delphi programmers can use COM classes built using C, and so forth. However, as you may be aware, COM's language independence is somewhat limited. For example, there is no way to derive a new COM class using
an existing COM class (as COM has no support for classical inheritance). Rather, you must make use of the more cumbersome "has-a" relationship to reuse COM class types.
Another benefit of COM is its location-transparent nature. Using constructs such as application identifiers (AppIDs), stubs, proxies, and the COM runtime environment,

programmers can avoid the need to work with raw sockets, RPC calls, and other low-level details. For example, consider the following
VB6 COM client code:

```
' This block of VB6 code can activate a COM class written in
' any COM-aware language, which may be located anywhere
' on the network (including your local machine).
Dim c as MyCOMClass
Set c = New MyCOMClass  ' Location resolved using AppID.
c.DoSomeWork
```

Although COM can be considered a very successful object model, it is extremely complex under the hood (at least until you have spent many months exploring its plumbing—especially if you happen to be a C++ programmer). To help simplify the development of COM binaries, numerous COM-aware frameworks have come into existence. For example, the Active Template Library (ATL) provides another set of C++ classes, templates, and macros to ease the creation of COM types. Many other languages also hide a good part of the COM infrastructure from view. However, language support alone is not enough to hide the complexity of COM. Even when you choose a relatively simply COM-aware language such as VB6, you are still forced to contend with fragile registration entries and numerous deployment-related issues (collectively termed *DLL hell*).

Life As a Windows DNA Programmer

To further complicate matters, there is a little thing called the Internet. Over the last several years,Microsoft has been adding more Internet-aware features into its family of operating systems and products. Sadly, building a web application using COM-based Windows Distributed interNet Applications Architecture (DNA) is also quite complex. Some of this complexity is due to the simple fact that Windows DNA requires the use of numerous technologies and languages (ASP, HTML, XML, JavaScript, VBScript, and COM(+), as well as a data access API such as ADO). One problem is that many of these technologies are completely unrelated from a syntactic point of view. For example, JavaScript has a syntax much like C, while VBScript is a subset of VB6. The COM servers that are created to run under the COM+ runtime have an entirely different look and feel from the ASP pages that invoke them. The result is a highly confused mishmash of technologies.

Furthermore, and perhaps more important, each language and/or technology has its own type system (that may look nothing like another's type system). An "int" in JavaScript is not quite the same as an "Integer" in VB6.

1.2 The .NET Solution

So much for the brief history lesson. The bottom line is that life as a Windows programmer has been tough. The .NET Framework is a rather radical and brute-force approach to making our lives easier. The solution proposed by .NET is "Change everything" (sorry, you can't blame the messenger for the message). As you will see during the remainder of this book, the .NET Framework is a completely new model for building systems on the Windows family of operating systems, as well as on numerous non-Microsoft operating systems such as Mac OS X and various Unix/Linux

distributions. To set the stage, here is a quick rundown of some core features provided courtesy of .NET:

• *Full interoperability with existing code*: This is (of course) a good thing. Existing COM binaries can commingle (i.e., interop) with newer .NET binaries and vice versa. Also, Platform Invocation Services (PInvoke) allows you to call C-based libraries (including the underlying API of the operating system) from .NET code.

• *Complete and total language integration*: Unlike COM, .NET supports cross-language inheritance, cross-language exception handling, and cross-language debugging.

• *A common runtime engine shared by all .NET-aware languages*: One aspect of this engine is a well-defined set of types that each .NET-aware language "understands."

• *A base class library*: This library provides shelter from the complexities of raw API calls and offers a consistent object model used by all .NET-aware languages.

• *No more COM plumbing*: `IClassFactory`, `IUnknown`, `IDispatch`, IDL code, and the evil `VARIANT` compliant data types (`BSTR`, `SAFEARRAY`, and so forth) have no place in a native .NET binary.

• *A truly simplified deployment model*: Under .NET, there is no need to register a binary unit
into the system registry. Furthermore, .NET allows multiple versions of the same `*.dll` to exist in harmony on a single machine. As you can most likely gather from the previous bullet points, the .NET platform has nothing to do with COM (beyond the fact that both frameworks originated from Microsoft). In fact, the only way .NET and COM types can interact with each other is using the interoperability layer.

1.3 Introducing the Building Blocks of the .NET Platform (the CLR, CTS, and CLS)

Now that you know some of the benefits provided by .NET, let's preview three key (and interrelated) entities that make it all possible: the CLR, CTS, and CLS. From a programmer's point of view, .NET can be understood as a new runtime environment and a comprehensive base class library. The run-time layer is properly referred to as the *common language runtime*, or *CLR*. The primary role of the CLR is to locate, load, and manage .NET types on your behalf. The CLR also takes care of a number of low-level details such as memory management and performing security checks. Another building block of the .NET platform is the *Common Type System*, or *CTS*. The CTS specification fully describes all possible data types and programming constructs supported by the runtime, specifies how these entities can interact with each other, and details how they are represented in the .NET metadata format (more information on metadata later in this chapter).

Understand that a given .NET-aware language might not support each and every feature defined by the CTS. The *Common Language Specification* (*CLS*) is a related specification that defines a subset of common types and programming constructs that all .NET programming languages can agree on. Thus, if you build .NET types that only expose CLS-compliant features, you can rest assured that all .NET-aware languages can consume them. Conversely, if you make use of a data type or programming construct that is outside of the bounds of the CLS, you cannot guarantee that every .NET program- ming language can interact with your .NET code library.

The Role of the Base Class Libraries

In addition to the CLR and CTS/CLS specifications, the .NET platform provides a base class library that is available to all .NET programming languages. Not only does this base class library encapsulate various primitives such as threads, file input/output (I/O), graphical rendering, and interaction with various external hardware devices, but it also provides support for a number of services required by most real-world applications. For example, the base class libraries define types that facilitate database access, XMLmanipulation, programmatic security, and the construction of web-enabled (as well as traditional desktop and console-based) front ends. From a high level, you can visualize the relationship between the CLR, CTS, CLS, and the base class library.

1.4 What C# Brings to the Table

Given that .NET is such a radical departure from previous technologies, Microsoft has developed a new programming language, C# (pronounced "see sharp"), specifically for this new platform. C# is a programming language that looks *very* similar (but not identical) to the syntax of Java. However, to call C# a Java rip-off is inaccurate. Both C# and Java are based on the syntactical constructs of C++. Just as Java is in many ways a cleaned-up version of C++, C# can be viewed as a cleaned-up version of Java—after all, they are all in the same family of languages.

The truth of the matter is that many of C#'s syntactic constructs are modeled after various aspects of Visual Basic 6.0 and C++. For example, like VB6, C# supports the notion of formal type properties (as opposed to traditional getter and setter methods) and the ability to declare methods taking varying number of arguments (via parameter arrays). Like C++, C# allows you to overload operators, as well as to create structures, enumerations, and callback functions (via delegates). Due to the fact that C# is a hybrid of numerous languages, the result is a product that is as syntactically clean—if not cleaner—than Java, is about as simple as VB6, and provides just about as much power and flexibility as C++ (without the associated ugly bits). In a nutshell, the C# language offers the following features (many of which are shared by other .NET-aware programming languages):

- No pointers required! C# programs typically have no need for direct pointer manipulation

(although you are free to drop down to that level if absolutely necessary).

- Automatic memory management through garbage collection. Given this, C# does not sup -port a `delete` keyword.
- Formal syntactic constructs for enumerations, structures, and class properties.
- The C++-like ability to overload operators for a custom type, without the complexity (e.g., making sure to "return *this to allow chaining" is not your problem).
- As of C# 2005, the ability to build generic types and generic members using a syntax very similar to C++ templates.
- Full support for interface-based programming techniques.
- Full support for aspect-oriented programming (AOP) techniques via attributes. This brand of development allows you to assign characteristics to types and their members to further qualify their behavior.

Perhaps the most important point to understand about the C# language shipped with the Microsoft .NET platform is that it can only produce code that can execute within the .NET runtime (you could never use C# to build a native COM server or a unmanaged Win32 API application). Officially speaking, the term used to describe the code targeting the .NET runtime is *managed code*. The binary unit that contains the managed code is termed an *assembly* (more details on assemblies in just a bit). Conversely, code that cannot be directly hosted by the .NET runtime is termed *unmanaged code*.

1.5 Additional .NET-Aware Programming Languages

Understand that C# is not the only language targeting the .NET platform. When the .NET platform was first revealed to the general public during the 2000 Microsoft Professional Developers Conference (PDC), several vendors announced they were busy building .NET-aware versions of their respective compilers. At the time of this writing, dozens of different languages have undergone .NET enlightenment. In addition to the five languages that ship with Visual Studio 2005 (C#, J#,Visual Basic .NET, Managed Extensions for C++, and JScript .NET), there are .NET compilers for Smalltalk, COBOL, and Pascal (to name a few). Although this book focuses (almost) exclusively on C#, Table 1-1 lists a number of .NET-enabled programming languages and where to learn more about them (do note that these URLs are subject to change).

Table 1-1. *A Sampling of .NET-Aware Programming Languages*

.NET Language Web Link	Meaning in Life
`http://www.oberon.ethz.ch/oberon.net`	Homepage for Active Oberon .NET.
`http://www.usafa.af.mil/df/dfcs/bios/`	Homepage for A# `mcc_html/a_sharp.cfm`
`http://www.netcobol.com`	For those interested in COBOL .NET.
`http://www.eiffel.com`	For those interested in Eiffel .NET.
`http://www.dataman.ro/dforth`	For those interested in Forth .NET.
`http://www.silverfrost.com/11/ftn95/ftn95_fortran_95_for_windows.asp`	For those interested in Fortran .NET.
`http://www.vmx-net.com`	Yes, even Smalltalk .NET is available.

Please be aware that Table 1-1 is not exhaustive. Numerous websites maintain a list of .NET-aware compilers, one of which would be `http://www.dotnetpowered.com/languages.aspx` (again, the exact URL is subject to change). I encourage you to visit this page, as you are sure to find many .NET languages worth investigating (LISP .NET, anyone?).

Life in a Multilanguage World

As developers first come to understand the language-agnostic nature of .NET, numerous questions arise. The most prevalent of these questions would have to be, "If all .NET languages compile down to 'managed code,' why do we need more than one compiler?" There are a number of ways to answer this question. First, we programmers are a *very* particular lot when it comes to our choice of programming language (myself included). Some of us prefer languages full of semicolons and curly brackets, with as few language keywords as possible. Others enjoy a language that offers more "human-readable" syntactic tokens (such as Visual Basic .NET). Still others may want to

leverage their mainframe skills while moving to the .NET platform (via COBOL .NET). Now, be honest. If Microsoft were to build a single "official" .NET language that was derived from the BASIC family of languages, can you really say all programmers would be happy with this choice? Or, if the only "official" .NET language was based on Fortran syntax, imagine all the folks out
there who would ignore .NET altogether. Because the .NET runtime couldn't care less which language was used to build a block of managed code, .NET programmers can stay true to their syntactic preferences, and share the compiled assemblies among teammates, departments, and external organizations (regardless of which .NET language others choose to use). Another excellent byproduct of integrating various .NET languages into a single unified software solution is the simple fact that all programming languages have their own sets of strengths and weak-nesses. For example, some programming languages offer excellent intrinsic support for advanced mathematical processing. Others offer superior support for financial calculations, logical calculations, interaction with mainframe computers, and so forth. When you take the strengths of a particular programming language and then incorporate the benefits provided by the .NET platform, everybody wins. Of course, in reality the chances are quite good that you will spend much of your time building software using your .NET language of choice. However, once you learn the syntax of one .NET language, it is very easy to master another. This is also quite beneficial, especially to the consultants of the world. If your language of choice happens to be C#, but you are placed at a client site that has committed to Visual Basic .NET, you should be able to parse the existing code body almost instantly (honest!) while still continuing to leverage the .NET Framework. Enough said.

1.6 An Overview of .NET Assemblies

Regardless of which .NET language you choose to program with, understand that despite the fact that .NET binaries take the same file extension as COM servers and unmanaged Win32 binaries (`*.dll` or `*.exe`), they have absolutely no internal. Perhaps most important, .NET binaries do not contain platform-specific instructions, but rather platform-agnostic *intermediate language* (*IL*) and type metadata. Figure 1-2 shows the big picture of the story thus far.

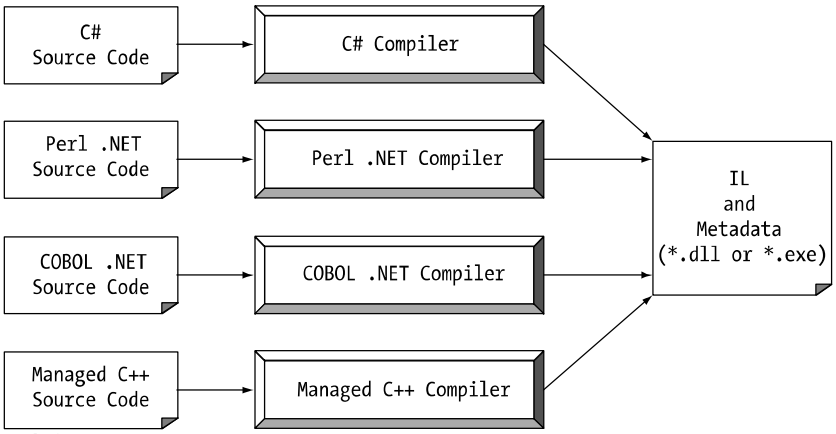


Figure 1-2. *All .NET-aware compilers emit IL instructions and metadata.*

Note There is one point to be made regarding the abbreviation "IL." During the development of .NET, the offi-
cial term for IL was Microsoft intermediate language (MSIL). However with the final release of .NET, the term was changed to common intermediate language (CIL). Thus, as you read the .NET literature, understand that IL, MSIL, and CIL are all describing the same exact entity. In keeping with the current terminology, I will use the abbreviation "CIL" throughout this text.

When a `*.dll` or `*.exe` has been created using a .NET-aware compiler, the resulting module is bundled into an *assembly*. You will examine numerous details of .NET assemblies in However, to facilitate the discussion of the .NET runtime environment, you do need to understand some basic properties of this new file format. As mentioned, an assembly contains CIL code, which is conceptually similar to Java bytecode in that it is not compiled to platform-specific instructions until absolutely necessary. Typically, "absolutely necessary" is the point at which a block of CIL instructions (such as a method imple-
mentation) is referenced for use by the .NET runtime.

In addition to CIL instructions, assemblies also contain *metadata* that describes in vivid detail the characteristics of every "type" living within the binary. For example, if you have a class named `SportsCar`, the type metadata describes details such as `SportsCar`'s base class, which interfaces are implemented by `SportsCar` (if any), as well as a full description of each member supported by the `SportsCar` type. .NET metadata is a dramatic improvement to COM type metadata. As you may already know, COM binaries are typically described using an associated type library (which is little more than a binary version of Interface Definition Language [IDL] code). The problems with COM type information are that it is not guaranteed to be present and the fact that IDL code has no way to document the externally referenced servers that are required for the correct operation of the current COM server. In contrast, .NET metadata is always present and is automatically generated by a given .NET-aware compiler.

Finally, in addition to CIL and type metadata,assemblies themselves are also described using metadata, which is officially termed a *manifest*. The manifest contains information about the current version of the assembly, culture information (used for localizing string and image resources), and a list of all externally referenced assemblies that are required for proper execution. You'll examine various tools that can be used to examine an assembly's types, metadata, and manifest information over the course of the next few chapters.

1.7 Single-File and Multifile Assemblies

In a great number of cases, there is a simple one-to-one correspondence between a .NET assembly and the binary file (`*.dll` or `*.exe`). Thus, if you are building a .NET `*.dll`, it is safe to cosider that the binary and the assembly are one and the same. Likewise, if you are building an executable desktop application, the `*.exe` can simply be referred to as the assembly itself. however, this is not completely accurate. Technically speaking, if an assembly is composed of a single `*.dll` or `*.exe` module, you have a *single-file assembly*. Single-file assemblies contain all the necessary CIL, metadata, and associated manifest in an autonomous, single, well-defined package.

Multifile assemblies, on the other hand, are composed of numerous .NET binaries, each of which is termed a *module*. When building a multifile assembly, one of these modules (termed the *primary module*) must contain the assembly manifest (and possibly CIL instructions and metadata for various types). The other related modules contain a module level manifest, CIL, and type metadata. As you might suspect, the primary module documents the set of required secondary modules within the assembly manifest. So, why would you choose to create a multifile assembly? When you partition an assembly into discrete modules, you end up with a more flexible deployment option. For example, if a user is referencing a remote assembly that needs to be downloaded onto his or her machine, the runtime will only download the required modules. Therefore, you are free to construct your assembly in such a way that less frequently required types (such as a type named `HardDriveReformatter`) are kept in a separate stand-alone module.
In contrast, if all your types were placed in a single-file assembly, the end user may end up downloading a large chunk of data that is not really needed (which is obviously a waste of time). Thus, as you can see, an assembly is really a *logical grouping* of one or more related modules that are intended to be initially deployed and versioned as a single unit.

1.8 The Role of the Common Intermediate Language

Now that you have a better feel for .NET assemblies, let's examine the role of the common intermediate language (CIL) in a bit more detail. CIL is a language that sits above any particular platform-specific instruction set. Regardless of which .NET-aware language you choose, the associated compiler emits CIL instructions. For example, the following C# code models a trivial calculator. Don't concern yourself with the exact syntax for now, but do notice the format of the `Add()` method in the `Calc` class:

```
// Calc.cs
using System;
namespace CalculatorExample
{
// This class contains the app's entry point.
public class CalcApp
{
static void Main()
{
Calc c = new Calc();
int ans = c.Add(10, 84);
Console.WriteLine("10 + 84 is {0}.", ans);

// Wait for user to press the Enter key before shutting down.
Console.ReadLine();
}
}
```

```
// The C# calculator.
public class Calc
{
public int Add(int x, int y)
{ return x + y; }
}
}
```

Once the C# compiler (csc.exe) compiles this source code file, you end up with a single-file *.exe assembly that contains a manifest, CIL instructions, and metadata describing each aspect of the Calc and CalcApp classes. For example, if you were to open this assembly using ildasm.exe (examined a little later in this chapter), you would find that the Add() method is represented using CIL such as the following:

```
.method public hidebysig instance int32 Add(int32 x, int32 y) cil managed
{
// Code size     8 (0x8)
.maxstack  2
.locals init ([0] int32 CS$1$0000)
IL_0000:  ldarg.1
IL_0001:  ldarg.2
IL_0002:  add
IL_0003:  stloc.0
IL_0004:  br.s     IL_0006
IL_0006:  ldloc.0
IL_0007:  ret
} // end of method Calc::Add
```

Don't worry if you are unable to make heads or tails of the resulting CIL for this method—will describe the basics of the CIL programming language. The point to concentrate on is that the C# compiler emits CIL, not platform-specific instructions. Now, recall that this is true of all .NET-aware compilers. To illustrate, assume you created this same application using Visual Basic .NET (VB .NET), rather than C#

```
' Calc.vb
Imports System

Namespace CalculatorExample
' A VB .NET 'Module' is a class that only contains
' static members.
Module CalcApp
Sub Main()
Dim ans As Integer
Dim c As New Calc
ans = c.Add(10, 84)
Console.WriteLine("10 + 84 is {0}.", ans)
Console.ReadLine()
End Sub
End Module
```

```
Class Calc
Public Function Add(ByVal x As Integer, ByVal y As Integer) As Integer
Return x + y
End Function
End Class
End Namespace
```

If you examine the CIL for the `Add()` method, you find similar instructions (slightly tweaked by
the VB .NET compiler):

```
.method public instance int32 Add(int32 x, int32 y) cil managed
{
// Code size     9 (0x9)
.maxstack  2
.locals init ([0] int32 Add)
IL_0000:  nop
IL_0001:  ldarg.1
IL_0002:  ldarg.2
IL_0003:  add.ovf
IL_0004:  stloc.0
IL_0005:  br.s     IL_0007
IL_0007:  ldloc.0
IL_0008:  ret
} // end of method Calc::Add
```

Benefits of CIL

At this point, you might be wondering exactly what is gained by compiling source code into CIL rather than directly to a specific instruction set. One benefit is language integration. As you have already seen, each .NET-aware compiler produces nearly identical CIL instructions. Therefore, all languages are able to interact within a well-defined binary arena. Furthermore, given that CIL is platform-agnostic, the .NET Framework itself is platform-agnostic, providing the same benefits Java developers have grown accustomed to (i.e., a single code base running on numerous operating systems). In fact, there is an international standard for the C# language, and a large subset of the .NET platform and implementations already exist for many non-Windows operating systems (more details at the conclusion of this chapter). In contrast to Java, however, .NET allows you to build applications using your language of choice.

Compiling CIL to Platform-Specific Instructions

Due to the fact that assemblies contain CIL instructions, rather than platform-specific instructions, CIL code must be compiled on the fly before use. The entity that compiles CIL code into meaningful CPU instructions is termed a *just-in-time* (*JIT*) *compiler*, which sometimes goes by the friendly name of *Jitter*. The .NET runtime environment leverages a JIT compiler for each CPU targeting the runtime,each optimized for the underlying platform. For example, if you are building a .NET application that is to be deployed to a handheld device (such as a Pocket PC), the corresponding Jitter is well equipped to run within low-memory environment. On the other hand, if you are deploying your assembly to a back-end server (where memory is seldom an issue),

the Jitter will be optimized to function in a high-memory environment. In this way, developers can write a single body of code that can be efficiently JIT-compiled and executed on machines with different architectures. Furthermore, as a given Jitter compiles CIL instructions into corresponding machine code, it will cache the results in memory in a manner suited to the target operating system. In this way, if a call is made to a method named `PrintDocument()`, the CIL instructions are compiled into platform-specific instructions on the first invocation and retained in memory for later use. Therefore, the next time `PrintDocument()` is called, there is no need to recompile the CIL.

1.9 The Role of .NET Type Metadata

In addition to CIL instructions, a .NET assembly contains full, complete, and accurate metadata, which describes each and every type (class, structure, enumeration, and so forth) defined in the binary, as well as the members of each type (properties, methods, events, and so on). Thankfully, it is always the job of the compiler (not the programmer) to emit the latest and greatest type meta- data. Because .NET metadata is so wickedly meticulous, assemblies are completely self-describing entities—so much so, in fact, that .NET binaries have no need to be registered into the system registry. To illustrate the format of .NET type metadata, let's take a look at the metadata that has been generated for the `Add()` method of the C# `Calc` class you examined previously (the metadata generated for the VB .NET version of the `Add()` method is similar):

```
TypeDef #2 (02000003)
-------------------------------------------------------
TypDefName: CalculatorExample.Calc  (02000003)
Flags    : [Public] [AutoLayout] [Class]
[AnsiClass] [BeforeFieldInit]  (00100001)
Extends  : 01000001 [TypeRef] System.Object
Method #1 (06000003)
-------------------------------------------------------
MethodName: Add (06000003)
Flags    : [Public] [HideBySig] [ReuseSlot]  (00000086)
RVA      : 0x00002090
ImplFlags : [IL] [Managed]  (00000000)
CallCnvntn: [DEFAULT]
hasThis
ReturnType: I4
2 Arguments
Argument #1:  I4
Argument #2:  I4
2 Parameters
(1) ParamToken : (08000001) Name : x flags: [none] (00000000)
(2) ParamToken : (08000002) Name : y flags: [none] (00000000)
```

Metadata is used by numerous aspects of the .NET runtime environment, as well as by various development tools. For example, the IntelliSense feature provided by Visual Studio

2005 is made possible by reading an assembly's metadata at design time. Metadata is also used by various object browsing utilities, debugging tools, and the C# compiler itself. To be sure, metadata is the backbone of numerous .NET technologies including remoting, reflection, late binding, XML web services, and object serialization.

1.10 The Role of the Assembly Manifest

Last but not least, remember that a .NET assembly also contains metadata that describes the assembly itself (technically termed a *manifest*). Among other details, the manifest documents all external assemblies required by the current assembly to function correctly, the assembly's version number, copyright information, and so forth. Like type metadata, it is always the job of the compiler to generate the assembly's manifest. Here are some relevant details of the `CSharpCalculator.exe` manifest:

```
.assembly extern mscorlib
{
.publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
.ver 2:0:0:0
}
.assembly CSharpCalculator
{
.hash algorithm 0x00008004
.ver 0:0:0:0
}
.module CSharpCalculator.exe
.imagebase 0x00400000
.subsystem 0x00000003
.file alignment 512
.corflags 0x00000001
```

In a nutshell, this manifest documents the list of external assemblies required by `CSharpCalculator.exe` (via the `.assembly extern` directive) as well as various characteristics of the assembly itself (version number, module name, and so on).

1.11 Understanding the Common Type System

A given assembly may contain any number of distinct "types." In the world of .NET, "type" is simply a generic term used to refer to a member from the set {class, structure, interface, enumeration, delegate}. When you build solutions using a .NET-aware language, you will most likely interact with each of these types. For example, your assembly may define a single class that implements some number of interfaces. Perhaps one of the interface methods takes an enumeration type as an input parameter and returns a structure to the caller. Recall that the Common Type System (CTS) is a formal specification that documents how types must be defined in order to be hosted by the CLR. Typically, the only individuals who are deeply concerned with the inner workings of the CTS are those building tools and/or compilersthat target the .NET platform. It is important, however, for all .NET programmers to learn about how

to work with the five types defined by the CTS in their language of choice. Here is a brief overview.

CTS Class Types

Every .NET-aware language supports, at the very least, the notion of a *class type*, which is the cornerstone of object-oriented programming (OOP). A class may be composed of any number of members (such as properties, methods, and events) and data points (fields). In C#, classes are declared using the `class` keyword:

```
// A C# class type.
public class Calc
{
public int Add(int x, int y)
{ return x + y; }
}
```

Table 1-2. *CTS Class Characteristics*

Class Characteristic	Meaning in Life
Is the class "sealed" or not?	Sealed classes cannot function as a base class to other classes.
Does the class implement any interfaces?	An *interface* is a collection of abstract members that provide a contract between the object and object user. The

CTS allows a class to implement any number of interfaces.

Is the class abstract or concrete? *Abstract* classes cannot be directly created, but are intended to define common behaviors for derived types. *Concrete* classes can be created directly. What is the "visibility" of this class? Each class must be configured with a visibility attribute. Basically, this trait defines if the class may be used by external
assemblies, or only from within the defining assembly (e.g., a private helper class).

CTS Structure Types

The concept of a structure is also formalized under the CTS. If you have a C background, you should be pleased to know that these user-defined types (UDTs) have survived in the world of .NET (although they behave a bit differently under the hood). Simply put, a *structure* can be thought of as a lightweight class type having value-based semantics. For more details on the subtleties of structures, see Chapter 3. Typically, structures are best suited for modeling geometric and mathematical data, and are created in C# using the `struct` keyword:

```
// A C# structure type.
struct Point
{
// Structures can contain fields.
public int xPos, yPos;

// Structures can contain parameterized constructors.
public Point(int x, int y)
{ xPos = x; yPos = y;}
```

```
// Structures may define methods.
public void Display()
{
Console.WriteLine("({0}, {1}", xPos, yPos);
}
```

CTS Interface Types

Interfaces are nothing more than a named collection of abstract member definitions, which may be supported (i.e., implemented) by a given class or structure. Unlike COM, .NET interfaces do *not* derive a common base interface such as `IUnknown`. In C#, interface types are defined using the `interface` key- word, for example:

```
// A C# interface type.
public interface IDraw
{
void Draw();
}
```

On their own, interfaces are of little use. However, when a class or structure implements a given interface in its unique way, you are able to request access to the supplied functionality using an interface reference in a polymorphic manner.

CTS Enumeration Types

Enumerations are a handy programming construct that allows you to group name/value pairs. For example, assume you are creating a video-game application that allows the player to select one of three character categories (Wizard, Fighter, or Thief). Rather than keeping track of raw numerical values to represent each possibility, you could build a custom enumeration using the `enum` keyword:

```
// A C# enumeration type.
public enum CharacterType
{
Wizard = 100,
Fighter = 200,
Thief = 300
}
```

By default, the storage used to hold each item is a 32-bit integer; however, it is possible to alter this storage slot if need be (e.g., when programming for a low-memory device such as a Pocket PC). Also, the CTS demands that enumerated types derive from a common base class, `System.Enum`. As you will see in Chapter 3, this base class defines a number of interesting members that allow you to extract, manipulate, and transform the underlying name/value pairs programmatically.

CTS Delegate Types

Delegates are the .NET equivalent of a type-safe C-style function pointer. The key difference is that a .NET delegate is a *class* that derives from `System.MulticastDelegate`, rather than a simple pointer to a raw memory address. In C#, delegates are declared using the `delegate` keyword:

```
// This C# delegate type can 'point to' any method
// returning an integer and taking two integers as input.
public delegate int BinaryOp(int x, int y);
```

Delegates are useful when you wish to provide a way for one entity to forward a call to another entity, and provide the foundation for the .NET event architecture. delegates have intrinsic support for multicasting (i.e., forwarding a request to multiple recipients) and
asynchronous method invocations.

CTS Type Members

Now that you have previewed each of the types formalized by the CTS, realize that most types take any number of *members*. Formally speaking, a *type member* is constrained by the set {constructor, finalizer, static constructor, nested type, operator, method, property, indexer, field, read only field, constant, event}. The CTS defines various "adornments" that may be associated with a given member. For example, each member has a given visibility trait (e.g., public, private, protected, and so forth). Some members may be declared as abstract to enforce a polymorphic behavior on derived types as well as virtual to define a canned (but overridable) implementation. Also, most members may be configured as static (bound at the class level) or instance (bound at the object level). The construction of
type members is examined over the course of the next several chapters.

Intrinsic CTS Data Types

The final aspect of the CTS to be aware of for the time being is that it establishes a well defined set of core data types. Although a given language typically has a unique keyword used to declare an intrinsic CTS data type, all language keywords ultimately resolve to the same type defined in an assembly named `mscorlib.dll`. Consider Table 1-3, which documents how key CTS data types are expressed in various .NET languages.

Table 1-3. *The Intrinsic CTS Data Types*

CTS Data Type	VB .NET Keyword	C# Keyword	Managed Extensions for C++ Keyword
System.Byte	Byte	byte	unsigned char
System.SByte	SByte	sbyte	signed char
System.Int16	Short	short	short
System.Int32	Integer	int	int or long
System.Int64	Long	long	__int64
System.UInt16	UShort	ushort	unsigned short
System.UInt32	UInteger	uint	unsigned int or unsigned long
System.UInt64	ULong	ulong	unsigned __int64
System.Single	Single	float	Float
System.Double	Double	double	Double
System.Object	Object	object	Object^
System.Char	Char	char	wchar_t
System.String	String	string	String^
System.Decimal	Decimal	decimal	Decimal
System.Boolean	Boolean	bool	Bool

1.12 Understanding the Common Language Specification

As you are aware, different languages express the same programming constructs in unique, language specific terms. For example, in C# you denote string concatenation

using the plus operator (+), while in VB .NET you typically make use of the ampersand (&). Even when two distinct languages express the same programmatic idiom (e.g., a function with no return value), the chances are very good that the syntax will appear quite different on the surface:

```
' VB .NET method returning nothing.
Public Sub MyMethod()
' Some interesting code...
End Sub

// C# method returning nothing.
public void MyMethod()
{
// Some interesting code...
}
```

As you have already seen, these minor syntactic variations are inconsequential in the eyes of the .NET runtime, given that the respective compilers (`vbc.exe` or `csc.exe`, in this case) emit a similar set of CIL instructions. However, languages can also differ with regard to their overall level of functionality. For example, a .NET language may or may not have a keyword to represent unsigned data, and may or may not support pointer types. Given these possible variations, it would be ideal to have a baseline to which all .NET-aware languages are expected to conform. The Common Language Specification (CLS) is a set of rules that describe in vivid detail the minimal and complete set of features a given .NET-aware compiler must support to produce code that can be hosted by the CLR, while at the same time be accessed in a uniform manner by all languages that target the .NET platform. In many ways, the CLS can be viewed as a *subset* of the full functionality defined by the CTS.

The CLS is ultimately a set of rules that compiler builders must conform to, if they intend their products to function seamlessly within the .NET universe. Each rule is assigned a simple name (e.g., "CLS Rule 6") and describes how this rule affects those who build the compilers as well as those who (in some way) interact with them. The crème de la crème of the CLS is the mighty Rule 1:

• *Rule 1*: CLS rules apply only to those parts of a type that are exposed outside the defining

assembly. Given this rule, you can (correctly) infer that the remaining rules of the CLS do not apply to the logic used to build the inner workings of a .NET type. The only aspects of a type that must conform to the CLS are the member definitions themselves (i.e., naming conventions, parameters, and return types). The implementation logic for a member may use any number of non-CLS techniques, as the outside world won't know the difference. To illustrate, the following `Add()` method is not CLS-compliant, as the parameters and return values make use of unsigned data (which is not a requirement of the CLS):

```
public class Calc
{
// Exposed unsigned data is not CLS compliant!
public ulong Add(ulong x, ulong y)
{ return x + y;}
```

```
}
```
However, if you were to simply make use of unsigned data internally as follows:
```
public class Calc
{
public int Add(int x, int y)
{
// As this ulong variable is only used internally,
// we are still CLS compliant.
ulong temp;
...
return x + y;
}
}
```
you have still conformed to the rules of the CLS, and can rest assured that all .NET languages are able to invoke the `Add()` method. Of course, in addition to Rule 1, the CLS defines numerous other rules. For example, the CLS describes how a given language must represent text strings, how enumerations should be represented internally (the base type used for storage), how to define static members, and so forth. Luckily, you don't have to commit these rules to memory to be a proficient .NET developer. Again, by and large, an intimate understanding of the CTS and CLS specifications is only of interest to tool/compiler builders.

Ensuring CLS Compliance

As you will see over the course of this book, C# does define a number of programming constructs that are not CLS-compliant. The good news, however, is that you can instruct the C# compiler to check your code for CLS compliance using a single .NET attribute:
```
// Tell the C# compiler to check for CLS compliance.
[assembly: System.CLSCompliant(true)]
```

1.13 The Assembly/Namespace/Type Distinction

Each of us understands the importance of code libraries. The point of libraries such as MFC, J2EE, and ATL is to give developers a well-defined set of existing code to leverage in their applications. However, the C# language does not come with a language-specific code library. Rather, C# developers leverage the language-neutral .NET libraries. To keep all the types within the base class libraries well organized, the .NET platform makes extensive use of the *namespace* concept. To clarify, Figure 1-4 shows a screen shot of the Visual Studio 2005 Object Brower utility. This tool allows you to examine the assemblies referenced by your current project, the namespaces within a particular assembly, the types within a given namespace, and the members of a specific type. Note that `mscorlib.dll` contains many different namespaces, each with its own semantically related types.

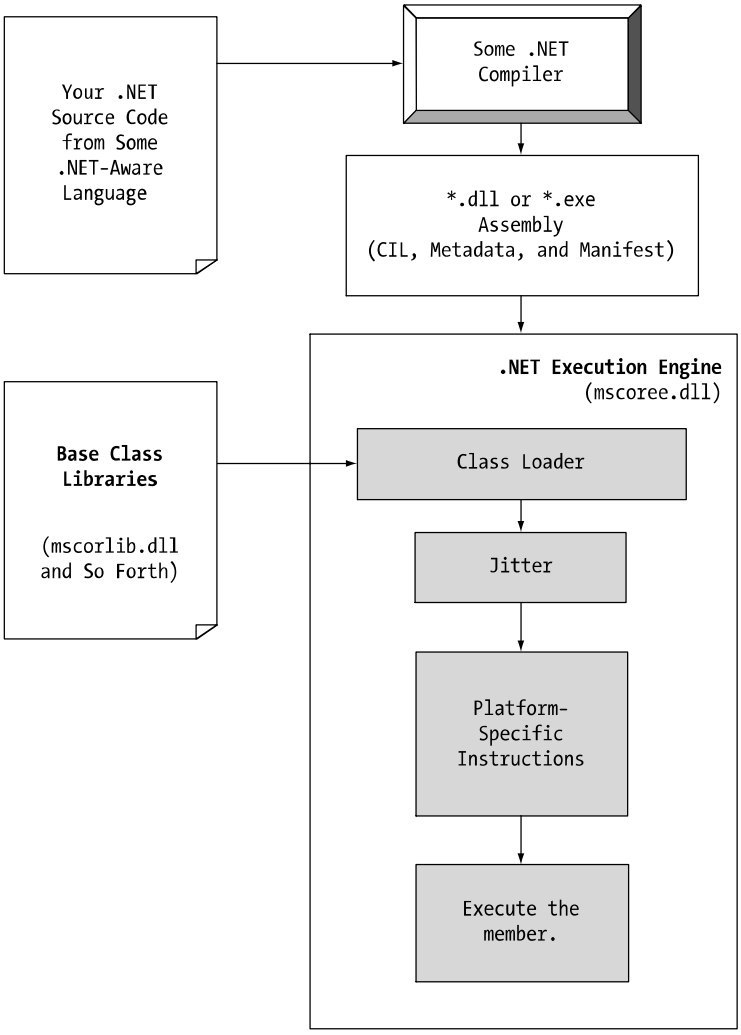


Figure 1-4. *A single assembly can have any number of namespaces.*

1.14 The Assembly/Namespace/Type Distinction

Each of us understands the importance of code libraries. The point of libraries such as MFC, J2EE, and ATL is to give developers a well-defined set of existing code to leverage in their applications. However, the C# language does not come with a language-specific code library. Rather, C# developers leverage the language-neutral .NET libraries. To keep all the types within the base class libraries well organized, the .NET platform makes extensive use of the *namespace* concept. Simply put, a namespace is a grouping of related types contained in an assembly. For example, the `System.IO` namespace contains file I/O related types, the `System.Data` namespace defines basic database types, and so on. It is very important to point out that a single assembly (such as `mscorlib.dll`) can contain any number of namespaces, each of which can contain any number of types.

Table 1-4. *A Sampling of .NET Namespaces*

.NET Namespace	Meaning in Life
`System`	Within `System` you find numerous useful types dealing with

	intrinsic data, mathematical computations, random number generation, environment variables, and garbage collection, as well as a number of commonly used exceptions and attributes.
`System.Collections`	These namespaces define a number of stock container objects
`System.Collections.Generic`	(`ArrayList`, `Queue`, and so forth), as well as base types and interfaces that allow you to build customized collections. As of .NET 2.0, the collection types have been extended with generic capabilities.
`System.Data`	These namespaces are used for interacting with databases using

1.15 Deploying the .NET Runtime

It should come as no surprise that .NET assemblies can be executed only on a machine that has the .NET Framework installed. As an individual who builds .NET software, this should never be an issue, as your development machine will be properly configured at the time you install the freely available .NET Framework 2.0 SDK (as well as commercial .NET development environments such as Visual Studio 2005). However, if you deploy an assembly to a computer that does not have .NET installed, it will fail to run. For this reason, Microsoft provides a setup package named `dotnetfx.exe` that can be freely shipped and installed along with your custom software. This installation program is included with the .NET Framework 2.0 SDK, and it is also freely downloadable from Microsoft. Once `dotnetfx.exe` is installed, the target machine will now contain the .NET base class libraries, .NET runtime (`mscoree.dll`), and additional .NET infrastructure (such as the GAC).

1.16 The Platform-Independent Nature of .NET

To close this chapter, allow me to briefly comment on the platform-independent nature of the .NET platform. To the surprise of most developers, .NET assemblies can be developed and executed on non-Microsoft operating systems (Mac OS X, numerous Linux distributions, BeOS, and FreeBSD, to name a few). To understand how this is possible, you need to come to terms to yet another abbreviation in the .NET universe: CLI (Common Language Infrastructure). When Microsoft released the C# programming language and the .NET platform, it also crafted a set of formal documents that described the syntax and semantics of the C# and CIL languages, the .NET assembly format, core .NET namespaces, and the mechanics of a hypothetical .NET runtime engine (known as the Virtual Execution System, or VES). Better yet, these documents have been submitted to Ecma International as official international standards (http://www.ecma-international.org). The specifications of interest are

- ECMA-334: The C# Language Specification

Recommended Questions:

1. Explain the builing block of .NET framework
2. Bring out the important differences between single and multifile assemblies
3. Whatarenamespaces? Listand explain the purpose of at least five namespaces.
4. What is .NET? With a neat diagram explain the important building blocks of .NET platform.

Building C# Applications
UNIT-2

2.1 The Role of the Command Line Complier (csc.exe)
2.2 Building C # Application using csc.exe
2.3 Working with csc.exe Response Files, Generating Bug Reports
2.4 The Command Line Debugger (cordbg.exe)
2.5 Using the, Visual Studio .NET IDE6
2.6 Other Key Aspects of the VS.NET IDE
2.7 C#"Preprocessor:" Directives
2.8 An Interesting Aside: The System. Environment Class.

2.1 The C# Command-Line Compiler (csc.exe)

There are a number of techniques you may use to compile C# source code. In addition to Visual Studio 2005 (as well as various third-party .NET IDEs), you are able to create .NET assemblies using the C# command-line compiler, `csc.exe` (where `csc` stands for *C-Sharp Compiler*). This tool is included with the .NET Framework 2.0 SDK. While it is true that you may never decide to build a large-scale application using the command-line compiler, it is important to understand the basics of how to compile your `*.cs` files by hand. I can think of a few reasons you should get a grip on the process:

- The most obvious reason is the simple fact that you might not have a copy of Visual Studio 2005.
- You plan to make use of automated build tools such as MSBuild or NAnt.
- You want to deepen your understanding of C#. When you use graphical IDEs to build applications, you are ultimately instructing `csc.exe` how to manipulate your C# input files. In this light, it's edifying to see what takes place behind the scenes. Another nice by-product of working with `csc.exe` in the raw is that you become that much more comfortable manipulating other command-line tools included with the .NET Framework 2.0 SDK. As you will see throughout this book, a number of important utilities are accessible only from the command line.

Configuring the C# Command-Line Compiler

Before you can begin to make use of the C# command-line compiler, you need to ensure that your development machine recognizes the existence of `csc.exe`. If your machine is not configured correctly, you are forced to specify the full path to the directory containing `csc.exe` before you can compile your C# files.

To equip your development machine to compile `*.cs` files from any directory, follow these steps (which assume a Windows XP installation; Windows NT/2000 steps will differ slightly):

1. Right-click the My Computer icon and select Properties from the pop-up menu.
2. Select the Advanced tab and click the Environment Variables button.
3. Double-click the Path variable from the System Variables list box.
4. Add the following line to the end of the current Path value (note each value in the Path variable is separated by a semicolon):

```
C:\Windows\Microsoft.NET\Framework\v2.0.50215
```

Of course, your entry may need to be adjusted based on your current version and location of the .NET Framework 2.0 SDK (so be sure to do a sanity check using Windows Explorer). Once you have updated the Path variable, you may take a test run by closing any command windows open in the background (to commit the settings), and then opening a new command window and entering

```
csc /?
```

If you set things up correctly, you should see a list of options supported by the C# compiler.

Configuring Additional .NET Command-Line Tools

Before you begin to investigate `csc.exe`, add the following additional Path variable to the System Variables list box (again, perform a sanity check to ensure a valid directory):

```
C:\Program Files\Microsoft Visual Studio 8\SDK\v2.0\Bin
```

Recall that this directory contains additional command-line tools that are commonly used during .NET development. With these two paths established, you should now be able to run any .NET utility from any command window. If you wish to confirm this new setting, close any open command windows, open a new command window, and enter the following command to view the options of the GAC utility, `gacutil.exe`:

```
gacutil /?
```

2.2 Building C# Applications Using csc.exe

Now that your development machine recognizes `csc.exe`, the next goal is to build a simple single file assembly named `TestApp.exe` using the C# command-line compiler and Notepad. First, you need some source code. Open Notepad and enter the following:

```
// A simple C# application.
using System;

class TestApp
{
public static void Main()
{
Console.WriteLine("Testing! 1, 2, 3");
}
}
```

Once you have finished, save the file in a convenient location (e.g., C:\CscExample) as `TestApp.cs`. Now, let's get to know the core options of the C# compiler. The first point of interest is to understand how to specify the name and type of assembly to create (e.g., a console application named `MyShell.exe`, a code library named `MathLib.dll`, a Windows Forms application named `MyWinApp.exe`, and so forth). Each possibility is represented by a specific flag passed into `csc.exe` as a command-line parameter (see Table 2-2).

Table 2-2. *Output-centric Options of the C# Compiler*

Option	Meaning in Life
`/out`	This option is used to specify the name of the assembly to be created. By default, the assembly name is the same as the name of the initial input `*.cs` file (in the case of a `*.dll`) or the name of the type containing the program's `Main()` method (in the case of an `*.exe`).
`/target:exe`	This option builds an executable console application. This is the default file output type, and thus may be omitted when building this application type.
`/target:library`	This option builds a single-file `*.dll` assembly.
`/target:module`	This option builds a *module*. Modules are elements of multifile assemblies (fully described in Chapter 11).

/target:winexe Although you are free to build Windows-based applications.

To compile TestApp.cs into a console application named TestApp.exe, change to the directory containing your source code file and enter the following command set (note that command-line flags must come before the name of the input files, not after):

```
csc /target:exe TestApp.cs
```

Here I did not explicitly specify an /out flag, therefore the executable will be named TestApp.exe,
given that TestApp is the class defining the program's entry point (the Main() method). Also be aware that most of the C# compiler flags support an abbreviated version, such as /t rather than /target (you can view all abbreviations by entering csc /? at the command prompt): csc /t:exe TestApp.cs Furthermore, given that the /t:exe flag is the default output used by the C# compiler, you could also compile TestApp.cs simply by typing csc TestApp.cs TestApp.exe can now be run from the command line.

Referencing External Assemblies

Next up, let's examine how to compile an application that makes use of types defined in a separate .NET assembly. Speaking of which, just in case you are wondering how the C# compiler understood your reference to the System.Console type, recall from Chapter 1 that mscorlib.dll is *automatically referenced* during the compilation process (if for some trange reason you wish to disable this behavior, you may specify the /nostdlib flag). To illustrate the process of referencing external assemblies, let's update the TestApp application to display a Windows Forms message box. Open your TestApp.cs file and modify it as follows:

```
using System;
// Add this!
using System.Windows.Forms;

class TestApp
{
public static void Main()
{
Console.WriteLine("Testing! 1, 2, 3");

// Add this!
MessageBox.Show("Hello...");
}
}
```

Notice the reference to the System.Windows.Forms namespace via the C# using keyword (introduced in Chapter 1). Recall that when you explicitly list the namespaces used within a given *.cs file, you avoid the need to make use of fully qualified names (which can lead to hand cramps). At the command line, you must inform csc.exe which assembly contains the "used" namespaces. Given that you have made use of the MessageBox class, you must specify the System.Windows. Forms.dll assembly using the /reference flag (which can be abbreviated to /r):

```
csc /r:System.Windows.Forms.dll testapp.cs
```

Compiling Multiple Source Files with csc.exe

The current incarnation of the `TestApp.exe` application was created using a single `*.cs` source code file. While it is perfectly permissible to have all of your .NET types defined in a single `*.cs` file, most projects are composed of multiple `*.cs` files to keep your code base a bit more flexible. Assume you have authored an additional class contained in a new file named `HelloMsg.cs`:

```
// The HelloMessage class
using System;
using System.Windows.Forms;

class HelloMessage
{
public void Speak()
{
MessageBox.Show("Hello...");
}
}
```

Now, update your initial `TestApp` class to make use of this new type, and comment out the previous Windows Forms logic:

```
using System;

// Don't need this anymore.
// using System.Windows.Forms;

class TestApp
{
public static void Main()
{
Console.WriteLine("Testing! 1, 2, 3");

// Don't need this anymore either.
// MessageBox.Show("Hello...");

// Exercise the HelloMessage class!
HelloMessage h = new HelloMessage();
h.Speak();
}
}
```

You can compile your C# files by listing each input file explicitly:

```
csc /r:System.Windows.Forms.dll testapp.cs hellomsg.cs
```

As an alternative, the C# compiler allows you to make use of the wildcard character (`*`) to inform `csc.exe` to include all `*.cs` files contained in the project directory as part of the current build:

```
csc /r:System.Windows.Forms.dll *.cs
```

When you run the program again, the output is identical. The only difference between the two applications is the fact that the current logic has been split among multiple files.

Referencing Multiple External Assemblies

On a related note, what if you need to reference numerous external assemblies using `csc.exe`? Simply list each assembly using a semicolon-delimited list. You don't need to specify multiple external assemblies for the current example, but some sample usage follows:

```
csc /r:System.Windows.Forms.dll;System.Drawing.dll *.cs
```

2.3 Working with csc.exe Response Files

As you might guess, if you were to build a complex C# application at the command prompt, your life would be full of pain as you type in the flags that specify numerous referenced assemblies and `*.cs` input files. To help lessen your typing burden, the C# compiler honors the use of *response files*. C# response files contain all the instructions to be used during the compilation of your current build. By convention, these files end in a `*.rsp` (response) extension. Assume that you have created a response file named `TestApp.rsp` that contains the following arguments (as you can see, comments are denoted with the `#` character):

```
# This is the response file
# for the TestApp.exe app
# of Chapter 2.

# External assembly references.
/r:System.Windows.Forms.dll

# output and files to compile (using wildcard syntax).
/target:exe /out:TestApp.exe *.cs
```

Now, assuming this file is saved in the same directory as the C# source code files to be compiled, you are able to build your entire application as follows (note the use of the `@` symbol): `csc @TestApp.rsp`

If the need should arise, you are also able to specify multiple `*.rsp` files as input (e.g., `csc @FirstFile.rsp @SecondFile.rsp @ThirdFile.rsp`). If you take this approach, do be aware that the compiler processes the command options as they are encountered! Therefore, command-line arguments in a later `*.rsp` file can override options in a previous response file. Also note that flags listed explicitly on the command line before a response file will be overridden by the specified `*.rsp` file. Thus, if you were to enter

```
csc /out:MyCoolApp.exe @TestApp.rsp
```

the name of the assembly would still be `TestApp.exe` (rather than `MyCoolApp.exe`), given the `/out:TestApp.exe` flag listed in the `TestApp.rsp` response file. However, if you list flags after a response file, the flag will override settings in the response file.

The Default Response File (csc.rsp)

The final point to be made regarding response files is that the C# compiler has an associated default response file (`csc.rsp`), which is located in the same directory as `csc.exe` itself (e.g., C:\Windows\Microsoft.NET\Framework\v2.0.50215) . If you were to open this file using Notepad, you will find that numerous .NET assemblies have already been specified using the `/r:` flag. When you are building your C# programs using `csc.exe`, this file will be automatically referenced, even when you supply a custom `*.rsp` file. Given the presence of the default response file, the current `TestApp.exe` application could be successfully compiled using the following command set (as `System.Windows.Forms.dll` is referenced within `csc.rsp`):

```
csc /out:TestApp.exe *.cs
```

In the event that you wish to disable the automatic reading of `csc.rsp`, you can specify the `/noconfig` option:

```
csc @TestApp.rsp /noconfig
```

Obviously, the C# command-line compiler has many other options that can be used to control how the resulting .NET assembly is to be generated. If you wish to learn more details regarding the functionality of `csc.exe`, look up my article titled "Working with the C# 2.0 Command Line Compiler" online at `http://msdn.microsoft.com`.

2.4 The Command-Line Debugger (cordbg.exe)

Before moving on to our examination of building C# applications using TextPad, I would like to briefly point out that the .NET Framework 2.0 SDK does ship with a command-line debugger named `cordbg.exe`. This tool provides dozens of options that allow you to debug your assembly. You may view them by specifying the `/?` flag:

```
cordbg /?
```

Table 2-3 documents some (but certainly not all) of the flags recognized by `cordbg.exe` (with the alternative shorthand notation) once you have entered a debugging session.

Table 2-3. *A Handful of Useful* `cordbg.exe` *Command-Line Flags*

Flag	Meaning in Life
`b[reak]`	Set or display current breakpoints.
`del[ete]`	Remove one or more breakpoints.
`ex[it]`	Exit the debugger.
`g[o]`	Continue debugging the current process until hitting next breakpoint.
`o[ut]`	Step out of the current function.
`p[rint]`	Print all loaded variables (local, arguments, etc.).
`si`	Step into the next line.
`so`	Step over the next line.

As I assume that most of you will choose to make use of the Visual Studio 2005 integrated debugger, I will not bother to comment on each flag of `cordbg.exe`. However,

for those of you who are interested, the following section presents a minimal walk-through of the basic process of debugging at the command line.

Debugging at the Command Line

Before you can debug your application using `cordbg.exe`, the first step is to generate debugging symbols for your current application by specifying the `/debug` flag of `csc.exe`. For example, to generate debugging data for `TestApp.exe`, enter the following command set:

```
csc @testapp.rsp /debug
```

This generates a new file named (in this case) `testapp.pdb`. If you do not have an associated `*.pdb` file, it is still possible to make use of `cordbg.exe`; however, you will not be able to view your C# source code during the process (which is typically no fun whatsoever, unless you wish to complicate matters by reading CIL code).

Once you have generated a `*.pdb` file, open a session with `cordbg.exe` by specifying your .NET assembly as a command-line argument (the `*.pdb` file will be loaded automatically):

```
cordbg.exe testapp.exe
```

When you wish to quit debugging with `cordbg.exe`, simply type `exit` (or the shorthand `ex`). Again, unless you are a command-line junkie, I assume you will opt for the graphical debugger provided by your IDE. If you require more information, look up `cordbg.exe` in the .NET Framework 2.0 SDK documentation.

2.5 Using The, Visual Studio .NET IDE

While Notepad is fine for creating simple .NET programs, it offers nothing in the way of developer productivity. It would be ideal to author `*.cs` files using an editor that supports (at a minimum) keyword coloring, code snippets, and integration with a C# compiler. As luck would have it, such a tool does exist: TextPad. TextPad is an editor you can use to author and compile code for numerous programming languages, including C#. The chief advantage of this product is the fact that it is very simple to use and provides just enough bells and whistles to enhance your coding efforts. To obtain TextPad, navigate to `http://www.textpad.com` and download the current version (4.7.3 at the time of this writing). Once you have installed the product, you will have a feature-complete version of TextPad; however, this tool is not freeware. Until you purchase a single-user license (for around US$30.00 at the time of this writing), you will be presented with a "friendly reminder" each time you run the application.

TextPad is not equipped to understand C# keywords or work with `csc.exe` out of the box. To do so, you will need to install an additional add-on. Navigate to

```
http://www.textpad.com/add-ons/syna2g.html
```

and download `csharp8.zip` using the "C# 2005" link option. This add-on takes into account the new keywords introduced with C# 2005 (in contrast to the "C#" link, which is limited to C# 1.1). Once you have unzipped `csharp8.zip`, place a copy of the extracted `csharp8.syn` file in the Samples subdirectory of the TextPad installation (e.g.,

C:\Program Files\TextPad 4\Samples). Next, launch TextPad and perform the following tasks using the New Document Wizard.

1. Activate the Configure New Document Class menu option.
2. Enter the name **C# 2.0** in the "Document class name" edit box.
3. In the next step, enter `*.cs` in the "Class members" edit box.
4. Finally, enable syntax highlighting, choose `csharp8.syn` from the drop-down list box, and

close the wizard. You can now tweak TextPad's C# support using the Document Classes node accessible from the Configure.

Configuring the *.cs File Filter

The next configuration detail is to create a filter for C# source code files displayed by the Open and Save dialog boxes:

1. Activate the Configure Preferences menu option and select File Name Filters from the

tree view control.

2. Click the New button, and enter **C#** into the Description field and `*.cs` into the Wild cards text box.
3. Move your new filter to the top of the list using the Move Up button and click OK.

Hooking Into csc.exe

The last major configuration detail to contend with is to associate `csc.exe` with TextPad so you can compile your C# files. The first way to do so is using the Tools Run menu option. Here you are presented with a dialog box that allows you to specify the name of the tool to run and any necessary command-line flags. To compile `TextPadTest.cs` into a .NET console-based executable, follow these steps:

1. Enter the full path to `csc.exe` into the Command text box (e.g., C:\Windows\Microsoft.NET\

Framework\v2.0. 50215\csc.exe).

2. Enter the command-line options you wish to specify within the Parameters text box (e.g.,

`/out:myApp.exe *.cs`). Recall that you can specify a custom response file to simplify matters

(e.g., `@myInput.rsp`)

3. Enter the directory containing the input files via the Initial folder text box (C:\TextPadTestApp in this example).
4. If you wish TextPad to capture the compiler output directly (rather than within a separate

command window), select the Capture Output check box.

At this point, you can either run your program by double-clicking the executable using Windows Explorer or leverage the Tools.

Associating Run Commands with Menu Items

TextPad also allows you to create custom menu items that represent predefined run commands. Let's create a custom item under the Tools menu named "Compile C# Console" that will compile all C# files in the current directory:

1. Activate the Configure Preferences menu option and select Tools from the tree view control.
2. Using the Add button, select Program and specify the full path to `csc.exe`.

3. If you wish, rename `csc.exe` to a more descriptive label (Compile C#) by clicking the tool
name and then clicking OK.
4. Finally, activate the Configure Preferences menu option once again, but this time select
Compile C# from the Tools node, and specify `*.cs` as the sole value in the Parameters field.
During the summer of 2004, Microsoft introduced a brand-new line of IDEs that fall under the designation of "Express" products (`http://msdn.microsoft.com/express`). To date, there are six members of the Express family:
• *Visual Web Developer 2005 Express*: A lightweight tool for building dynamic websites and
XML web services using ASP.NET 2.0
• *Visual Basic 2005 Express*: A streamlined programming tool ideal for novice programmers who want to learn how to build applications using the user-friendly syntax of Visual Basic .NET *Visual C# 2005 Express, Visual C++ 2005 Express, and Visual J# 2005 Express*: Targeted IDEs for students and enthusiasts who wish to learn the fundamentals of computer science in their syntax of choice
• *SQL Server 2005 Express*: An entry-level database management system geared toward hobbyists, enthusiasts, and student developers

2.6 Other Key Aspects of the VS.NET IDE

Visual Studio 2005 gives us the ability to design classes visually (but this capability is not included in Visual C# 2005 Express). The Class Designer utility allows you to view and modify the relationships of the types (classes, interfaces, structures, enumerations, and delegates) in your project. Using this tool, you are able to visually add (or remove) members to (or from) a type and have your modifications reflected in the corresponding C# file. As well, as you modify a given C# file, changes are reflected in the class diagram.
To work with this aspect of Visual Studio 2005, the first step is to insert a new class diagram file. There are many ways to do so, one of which is to click the View Class Diagram button located on Solution Explorer's right side

2.7 C# Preprocessor Directives

Like many other languages in the C family, C# supports the use of various symbols that allow you to interact with the compilation process. Before examining various C# preprocessor directives, let's get our terminology correct. The term "C# preprocessor directive" is not entirely accurate. In reality, this term is used only for consistency with the C and C++ programming languages. In C#, there is no separate preprocessing step. Rather, preprocessing directives are processed as part of the lexical analysis phase of the compiler.
In any case, the syntax of the C# preprocessor directives is very similar to that of the other mem bers of the C family, in that the directives are always prefixed with the pound sign (#). Table 9-4 defines some of the more commonly used directives (consult the .NET Framework 2.0 SDK documentation for complete details).
Table 9-4. *Common C# Preprocessor Directives*

Directives	Meaning in Life

`#region, #endregion`	Used to mark sections of collapsible source code
`#define, #undef`	Used to define and undefine conditional compilation symbols
`#if, #elif, #else, #endif`	Used to conditionally skip sections of source code (based on specified compilation symbols)

Specifying Code Regions

Perhaps some of the most useful of all preprocessor directives are `#region` and `#endregion`. Using these tags, you are able to specify a block of code that may be hidden from view and identified by a friendly textual marker. Use of regions can help keep lengthy `*.cs` files more manageable. For example, you could create one region for a type's constructors, another for type properties, and so forth:

```
class Car
{
private string petName;
private int currSp;

#region Constructors
public Car()
{ ... }
public Car Car(int currSp, string petName)
{...}
#endregion

#region Properties
public int Speed
{ ... }
public string Name
{...}
#endregion
}
```

2.8 An Interesting Aside: The System.Environment Class

Let's examine the `System.Environment` class in greater detail. This class allows you to obtain a number of details regarding the operating system currently hosting your .NET application using various static members. To illustrate this class's usefulness, update your `Main()` method with the following logic:

```
public static int Main(string[] args)
{
...
// OS running this app?
Console.WriteLine("Current OS: {0} ", Environment.OSVersion);

// Directory containing this app?
Console.WriteLine("Current Directory: {0} ",
Environment.CurrentDirectory);
```

```
// List the drives on this machine.
string[] drives = Environment.GetLogicalDrives();
for(int i = 0; i < drives.Length; i++)
Console.WriteLine("Drive {0} : {1} ", i, drives[i]);

// Which version of the .NET platform is running this app?
Console.WriteLine("Executing version of .NET: {0} ",
Environment.Version);
...
}
```

The `System.Environment` type defines members other than those presented in the previous example. Table 3-1 documents some additional properties of interest; however, be sure to check out the .NET Framework 2.0 SDK documentation for full details.

Table 3-1. *Select Properties of* `System.Environment`

Property	Meaning in Life
`MachineName`	Gets the name of the current machine
`NewLine`	Gets the newline symbol for the current environment
`ProcessorCount`	Returns the number of processors on the current machine
`SystemDirectory`	Returns the full path to the system directory
`UserName`	Returns the name of the entity that started this application

Recommended Questions:

1. Expalin with a neat diagram the workflow that takes place between your source code, a given .NET compiler and the .NET execution engine.
2. How .NET framework different from other programming environments like COM,C++,VB6,java etc?
3. Briefly discuss the state of affairs that eventually led to the .NET platform. What is the .NET solution and what C# brings t the tables?
4. What are the basic building blocks of .NET platform? Explain the common type system in detail.

UNIT-3
The C# Programming Language

3.1 The Anatomy of a Basic C# Class
3.2 Creating objects: Constructor Basics
3.3 The Composition of a C# Application
3.4 Default Assignment and Variable Scope
3.5 The C# Member Initialization Syntax
3.6 Basic Input and Output with the Console Class
3.7 Understanding Value Types and Reference Types
3.8 The Master Node: System, Object
3.9 The System Data Types (and C# Aliases)
3.10 Converting Between Value Types and Reference Types: Boxing and Unboxing,
3.11 Defining Program Constants
3.12 C# Iteration Constructs
3.13 C# Controls Flow Constructs
3.14 The Complete Set of C
3.15 Methods Parameter Modifies
3.16 Array Manipulation in C #
3.17 String Manipulation in C#
3.18 C# Enumerations
3.19 Defining Structures in C#
3.20 Defining Custom Namespaces.

3.1 The Anatomy of a Simple C# Program

C# demands that all program logic is contained within a type definition (recall from Chapter 1 that *type* is a term referring to a member of the set {class, interface, structure, enumeration, delegate}). Unlike in C(++), in C# it is not possible to create global functions or global points of data. In its simplest form, a C# program can be written as follows:

```
// By convention, C# files end with a *.cs file extension.
using System;

class HelloClass
{
public static int Main(string[] args)
{
Console.WriteLine("Hello World!");
Console.ReadLine();
return 0;
}
}
```

Here, a definition is created for a class type (`HelloClass`) that supports a single method named `Main()`. Every executable C# application must contain a class defining a `Main()` method, which is used to signify the entry point of the application. As you can see, this signature of `Main()` is adorned with the `public` and `static` keywords. Later in this chapter, you will be supplied with a formal definition of "public" and "static." Until then, understand that public members are accessible from other types, while static members are scoped at the class level (rather than the object level) and can thus be invoked without the need to first create a new class instance.

Variations on the Main() Method

The previous iteration of `Main()` was defined to take a single parameter (an array of strings) and return an integer data type. This is not the only possible form of `Main()`, however. It is permissible to construct your application's entry point using any of the following signatures (assuming it is contained within a C# class or structure definition):

```
// No return type, array of strings as argument.
public static void Main(string[] args)
{
}

// No return type, no arguments.
public static void Main()
{
}

// Integer return type, no arguments.
public static int Main()
{
}
```

Processing Command-Line Arguments

Assume that you now wish to update `HelloClass` to process possible command-line parameters:

```
// This time, check if you have been sent any command-line arguments.
using System;

class HelloClass{
public static int Main(string[] args)
{
Console.WriteLine("***** Command line args *****");
for(int i = 0; i < args.Length; i++)
Console.WriteLine("Arg: {0} ", args[i]);
...
}
}
```

Here, you are checking to see if the array of strings contains some number of items using the `Length` property of `System.Array` (as you'll see later in this chapter, all C# arrays actually alias the `System.Array` type, and therefore have a common set of members). As you loop over each item in the array, its value is printed to the console window. Supplying the arguments at the command line is equally as simple. As an alternative to the standard `for` loop, you may iterate over incoming string arrays using the C# `foreach` keyword. This bit of syntax is fully explained later in this chapter, but here is some sample usage:

```
// Notice you have no need to check the size of the array when using
'foreach'.
public static int Main(string[] args)
{
...
foreach(string s in args)
Console.WriteLine("Arg: {0} ", s);
...
}
```

Finally, you are also able to access command-line arguments using the static `GetCommand LineArgs()` method of the `System.Environment` type. The return value of this method is an array of strings. The first index identifies the current directory containing the application itself, while the remaining elements in the array contain the individual command-line arguments (when using this technique, you no longer need to define the `Main()` method as taking a string array parameter)

```
public static int Main(string[] args)
{
...
```

```
// Get arguments using System.Environment.
string[] theArgs = Environment.GetCommandLineArgs();
Console.WriteLine("Path to this app is: {0}", theArgs[0]);
...
}
```

3.2 Creating Objects:Constructor Basics

Now that you have the role of `Main()` under your belt, let's move on to the topic of object construction. All object-oriented (OO) languages make a clear distinction between classes and objects. A *class* is a definition (or, if you will, a *blueprint*) for a user-defined type (UDT). An *object* is simply a term describing a given instance of a particular class in memory. In C#, the `new` keyword is the de facto way to create an object. Unlike other OO languages (such as C++), it is not possible to allocate a class type on the stack; therefore, if you attempt to use a class variable that has not been "new-ed," you are issued a compile-time error. Thus the following C# code is *illegal*:

```
using System;
class HelloClass
{
public static int Main(string[] args)
{
// Error!  Use of unassigned local variable! Must use 'new'.
HelloClass c1;
c1.SomeMethod();
...
}
}
```

To illustrate the proper procedures for object creation, observe the following update:

```
using System;

class HelloClass
{
public static int Main(string[] args)
{
// You can declare and create a new object in a single line...
HelloClass c1 = new HelloClass();

// ...or break declaration and creation into two lines.
HelloClass c2;
c2 = new HelloClass();
...
}
}
```

The `new` keyword is in charge of calculating the correct number of bytes for the specified object and acquiring sufficient memory from the managed heap. Here, you have allocated two objects of the `HelloClass` class type. Understand that C# object variables are really a *reference* to the object in memory, not the actual object itself. Thus, in this

light, c1 and c2 each reference a unique HelloClass object allocated on the managed heap.

3.3 The Composition of a C#

The previous HelloClass objects have been constructed using the *default constructor*, which by definition never takes arguments. Every C# class is automatically provided with a free default constructor, which you may redefine if need be. The default constructor ensures that all member data is set to an appropriate default value (this behavior is true for all constructors). Contrast this to C++, where unini- tialized state data points to garbage (sometimes the little things mean a lot). Typically, classes provide additional constructors beyond the default. In doing so, you provide the object user with a simple way to initialize the state of an object at the time of creation. Like in Java and C++, in C# constructors are named identically to the class they are constructing, and they never provide a return value (not even void). Here is the HelloClass type once again, with a custom constructor, a redefined default constructor, and a point of public string data:

```
// HelloClass, with constructors.
using System;

class HelloClass
{
// A point of state data.
// Default constructor.
public HelloClass()
{ Console.WriteLine("Default ctor called!"); }

// This custom constructor assigns state data
// to a user-supplied value.
public HelloClass (string msg)
{
Console.WriteLine("Custom ctor called!");
userMessage = msg;
}

// Program entry point.
public static int Main(string[] args)
{
// Call default constructor.
HelloClass c1 = new HelloClass();
Console.WriteLine("Value of userMessage: {0}\n", c1.userMessage);

// Call parameterized constructor.
HelloClass c2;
c2 = new HelloClass("Testing, 1, 2, 3");
Console.WriteLine("Value of userMessage: {0}", c2.userMessage);
Console.ReadLine();
```

```
return 0;
}
}
```

3.4 Default Assignment and variable scope

Currently, the `HelloClass` type has been constructed to perform two duties. First, the class defines the entry point of the application (the `Main()` method). Second, `HelloClass` maintains a point of field data and a few constructors. While this is all well and good, it may seem a bit strange (although syntactically well-formed) that the static `Main()` method creates an instance of the very class in which it was defined:

```
class HelloClass
{
...
public static int Main(string[] args)
{
HelloClass c1 = new HelloClass();
...
}
}
```

Many of my initial examples take this approach, just to keep focused on illustrating the task at hand. However, a more natural design would be to refactor the current `HelloClass` type into two distinct classes: `HelloClass` and `HelloApp`. When you build C# applications, it becomes quite common to have one type functioning as the "application object" (the type that defines the `Main()` method) and numerous other types that constitute the application at large In OO parlance, this is termed the *separation of concerns*. In a nutshell, this design principle states that a class should be responsible for the least amount of work. Thus, we could reengineer the current program into the following (notice that a new member named `PrintMessage()` has been added to the `HelloClass` type):

```
class HelloClass
{
public string userMessage;

public HelloClass()
{ Console.WriteLine("Default ctor called!"); }

public HelloClass(string msg)
{
Console.WriteLine("Custom ctor called!");
userMessage = msg;
}

public void PrintMessage()
{
Console.WriteLine("Message is: {0}", userMessage);
```

```
}
}

class HelloApp
{
public static int Main(string[] args)
{
HelloClass c1 = new HelloClass("Hey there...");
c1.PrintMessage();
...
}
}
```

3.5 C# Member Intialization Syntax

Many of the example applications created over the course of these first few chapters make extensive use of the `System.Console` class. While a console user interface (CUI) is not as enticing as a Windows or web UI, restricting the early examples to a CUI will allow us to keep focused on the concepts under examination, rather than dealing with the complexities of building GUIs. As its name implies, the `Console` class encapsulates input, output, and error stream manipulations for console-based applications. With the release of .NET 2.0, the `Console` type has been enhanced with additional functionality. Table 3-2 lists some (but not all) new members of interest.

Table 3-2. *Select .NET 2.0–Specific Members of* `System.Console`

Member	Meaning in Life
`BackgroundColor`	These properties set the background/foreground colors for the current output. They may be assigned any member of the `ConsoleColor` enumeration.
`ForegroundColor`	
`BufferHeight`	These properties control the height/width of the console's buffer area.
`BufferWidth`	
`Clear()`	This method clears the buffer and console display area.
`Title`	This property sets the title of the current console.
`WindowHeight`	These properties control the dimensions of the console in relation to the established buffer.
`WindowWidth`	
`WindowTop`	
`WindowLeft`	

3.6 Basic Input and Output with the Console Class

In addition to the members in Table 3-2, the `Console` type defines a set of methods to capture input and output, all of which are defined as static and are therefore called at

the class level. As you have seen, `WriteLine()` pumps a text string (including a carriage return) to the output stream. The `Write()` method pumps text to the output stream without a carriage return. `ReadLine()` allows you to receive information from the input stream up until the carriage return, while `Read()` is used to capture a single character from the input stream. To illustrate basic I/O using the `Console` class, consider the following `Main()` method, which prompts the user for some bits of information and echoes each item to the standard output stream.
Figure 3-5 shows a test run.

```
// Make use of the Console class to perform basic I/O.
static void Main(string[] args)
{
// Echo some stats.
Console.Write("Enter your name: ");
string s = Console.ReadLine();
Console.WriteLine("Hello, {0} ", s);

Console.Write("Enter your age: ");
s = Console.ReadLine();
Console.WriteLine("You are {0} years old", s);
}
```

3.7 Understanding Value Types and Reference Types

Like any programming language, C# defines a number of keywords that represent basic data types such as whole numbers, character data, floating-point numbers, and Boolean values. If you come from a C++ background, you will be happy to know that these intrinsic types are fixed constants in the universe, meaning that when you create an integer data point, all .NET-aware languages understand the fixed nature of this type, and all agree on the range it is capable of handling.
Specifically speaking, a .NET data type may be *value-based* or *reference-based*. Value-based types, which include all numerical data types (`int`, `float`, etc.), as well as enumerations and structures, are allocated *on the stack*. Given this factoid, value types can be quickly removed from memory once they fall out of the defining scope:

```
// Integers are value types!
public void SomeMethod()
{
int i = 0;
Console.WriteLine(i);
} // 'i' is popped off the stack here!
```

When you assign one value type to another, a member-by-member copy is achieved by default. In terms of numerical or Boolean data types, the only "member" to copy is the value of the variable itself:

```
// Assigning two intrinsic value types results in
// two independent variables on the stack.
public void SomeMethod()
{
int i = 99;
```

```
int j = i;

// After the following assignment, 'i' is still 99.
j = 8732;
}
```

While the previous example is no major newsflash, understand that .NET structures (and enumerations, which are examined later in this chapter) are also value types. Structures, simply put, provide a way to achieve the bare-bones benefits of object orientation (i.e., encapsulation) while having the efficiency of stack-allocated data. Like a class, structures can take constructors (provided they have arguments) and define any number of members. All structures are implicitly derived from a class named `System.ValueType`. Functionally, the only purpose of `System.ValueType` is to "override" the virtual methods defined by `System.Object` (described in just a moment) to honor value-based, versus reference-based, semantics. In fact, the instance methods defined by `System.ValueType` are identical to those of `System.Object`:

```
// Structures and enumerations extend System.ValueType.
public abstract class ValueType : object
{
public virtual bool Equals(object obj);
public virtual int GetHashCode();
public Type GetType();
public virtual string ToString();
}
```

Assume you have created a C# structure named `MyPoint`, using the C# `struct` keyword:

```
// Structures are value types!
struct MyPoint
{
public int x, y;
}
```

To allocate a structure type, you may make use of the `new` keyword, which may seem counterintuitive given that we typically think `new` always implies heap allocation. This is part of the smoke and mirrors maintained by the CLR. As programmers, we can assume everything is an object and `new` value types. However, when the runtime encounters a type derived from `System.ValueType`, stack allocation is achieved:

```
// Still on the stack!
MyPoint p = new MyPoint();
```

As an alternative, structures can be allocated without using the `new` keyword:

```
MyPoint p1;
p1.x = 100;
p1.y = 100;
```

If you take this approach, however, you *must* initialize each piece of field data before use.

Failing to do so results in a compiler error.

Value Types, References Types, and the Assignment Operator

```
static void Main(string[] args)
{
```

```
Console.WriteLine("***** Value Types / Reference Types *****");
Console.WriteLine("-> Creating p1");
MyPoint p1 = new MyPoint();
p1.x = 100;
p1.y = 100;
Console.WriteLine("-> Assigning p2 to p1\n");
MyPoint p2 = p1;

// Here is p1.
Console.WriteLine("p1.x = {0}", p1.x);
Console.WriteLine("p1.y = {0}", p1.y);

// Here is p2.
Console.WriteLine("p2.x = {0}", p2.x);
Console.WriteLine("p2.y = {0}", p2.y);

// Change p2.x. This will NOT change p1.x.
Console.WriteLine("-> Changing p2.x to 900");
p2.x = 900;

// Print again.
Console.WriteLine("-> Here are the X values again...");
Console.WriteLine("p1.x = {0}", p1.x);
Console.WriteLine("p2.x = {0}", p2.x);
Console.ReadLine();
}
// Classes are always reference types.
class MyPoint // <= Now a class!
{
public int x, y;
}
```

Value Types Containing Reference Types

Now that you have a better feeling for the differences between value types and reference types, let's examine a more complex example. Assume you have the following reference (class) type that maintains an informational string that can be set using a custom constructor:

```
class ShapeInfo
{
public string infoString;
public ShapeInfo(string info)
{ infoString = info; }
}
```

Now assume that you want to contain a variable of this class type within a value type named `MyRectangle`. To allow the outside world to set the value of the inner `ShapeInfo`, you also provide a custom constructor (as explained in just a bit, the default constructor of a structure is reserved and cannot be redefined):

```
struct MyRectangle
{
// The MyRectangle structure contains a reference type member.
public ShapeInfo rectInfo;

public int top, left, bottom, right;
public MyRectangle(string info)
{
rectInfo = new ShapeInfo(info);
top = left = 10;
bottom = right = 100;
}
}
```

At this point, you have contained a reference type within a value type. The million-dollar question now becomes, what happens if you assign one `MyRectangle` variable to another? Given what you already know about value types, you would be correct in assuming that the integer data (which is indeed a structure) should be an independent entity for each `MyRectangle` variable.

```
static void Main(string[] args)
{
// Create the first MyRectangle.
Console.WriteLine("-> Creating r1");
MyRectangle r1 = new MyRectangle("This is my first rect");

// Now assign a new MyRectangle to r1.
Console.WriteLine("-> Assigning r2 to r1");
MyRectangle r2;
r2 = r1;

// Change values of r2.
Console.WriteLine("-> Changing all values of r2");
r2.rectInfo.infoString = "This is new info!";
r2.bottom = 4444;

// Print values
Console.WriteLine("-> Values after change:");
Console.WriteLine("-> r1.rectInfo.infoString: {0}", r1.rectInfo.infoString);
Console.WriteLine("-> r2.rectInfo.infoString: {0}", r2.rectInfo.infoString);
Console.WriteLine("-> r1.bottom: {0}", r1.bottom);
Console.WriteLine("-> r2.bottom: {0}", r2.bottom);
}
```

Value and Reference Types: Final Details

To wrap up this topic, ponder the information in Table 3-8, which summarizes the core distinctions between value types and reference types.

Table 3-8. *Value Types and Reference Types Side by Side*

Intriguing Question	Value Type	Reference Type
Where is this type allocated?	Allocated on the stack.	Allocated on the managed heap.
How is a variable represented?	Value type variables are local	Reference type variables are copies.pointing to the memory occupied by the allocated instance.
What is the base type?	Must derive from	Can derive from any other type

3.8 The Master Node: System.Object

In .NET, every type is ultimately derived from a common base class: `System.Object`. The `Object` class defines a common set of members supported by every type in the .NET universe. When you create a class that does not explicitly specify its base class, you implicitly derive from `System.Object`:

```
// Implicitly deriving from System.Object.
class HelloClass
{...}
```

If you wish to be more clear with your intension, the C# colon operator (:) allows you to explicitly specify a type's base class (such as `System.Object`):

```
// In both cases we are explicitly deriving from System.Object.
class ShapeInfo : System.Object
{...}

class ShapeInfo : object
{...}
```

`System.Object` defines a set of instance-level and class-level (static) members. Note that some of the instance-level members are declared using the `virtual` keyword and can therefore be *overridden* by a derived class:

```
// The topmost class in the .NET universe: System.Object
namespace System
{
public class Object
{
public Object();
public virtual Boolean Equals(Object obj);
public virtual Int32 GetHashCode();
public Type GetType();
public virtual String ToString();
protected virtual void Finalize();
protected Object MemberwiseClone();
public static bool Equals(object objA, object objB);
public static bool ReferenceEquals(object objA, object objB);
}
}
```

The Default Behavior of System.Object

To illustrate some of the default behavior provided by the `System.Object` base class, assume a class named `Person` defined in a custom namespace named `ObjectMethods`:

```
// The 'namespace' keyword is fully examined at the end of this chapter.
namespace ObjectMethods
{
class Person
{
public Person(string fname, string lname, string s, byte a)
{
firstName = fname;
lastName = lname;
SSN = s;
age = a;
}
public Person(){}

// The state of a person.
public string firstName;public string lastName;
public string SSN;
public byte age;
}
}
```

Now, within our `Main()` method, we make use of the `Person` type as so:

```
static void Main(string[] args)
{
Console.WriteLine("***** Working with Object *****\n");

Person fred = new Person("Fred", "Clark", "111-11-1111", 20);
Console.WriteLine("-> fred.ToString: {0}", fred.ToString());
Console.WriteLine("-> fred.GetHashCode: {0}", fred.GetHashCode());
// Make some other references to 'fred'.
Person p2 = fred;
object o = p2;

// Are all 3 instances pointing to the same object in memory?
if(o.Equals(fred) && p2.Equals(o))
Console.WriteLine("fred, p2 and o are referencing the same object!");
Console.ReadLine();
}
```

First, notice how the default implementation of `ToString()` simply returns the fully qualified name of the type (e.g., `namespace.typename`). `GetType()` retrieves a `System.Type` object, which defines a property named `BaseType` (as you can guess, this will identify the fully qualified name of the type's base class). Now, reexamine the code that leverages the `Equals()` method. Here, a new `Person` object is placed on the managed

heap, and the reference to this object is stored in the `fred` reference variable. `p2` is also of type `Person`, however, you are not creating a *new* instance of the `Person` class, but assigning `p2` to `fred`. Therefore, `fred` and `p2` are both pointing to the same object in memory, as is the variable `o` (of type `object`, which was thrown in for good measure). Given that `fred`, `p2`, and `o` all point to the same object in memory, the equality test succeeds.

Overriding Some Default Behaviors of
System.Object Although the canned behavior of `System.Object` can fit the bill in most cases, it is quite common for your custom types to override some of these inherited methods. Chapter 4 provides a complete examination of OOP under C#, but in a nutshell, *overriding* is the process of redefining the behavior of an inherited *virtual* member in a derived class. As you have just seen, `System.Object` defines a number of virtual methods (such as= `ToString()` and `Equals()`) that do define a canned implementtation. However, if you want to build a custom implementation of these virtual members for a derived type, you make use of the C# `override` keyword.

Overriding System.Object.ToString()
Overriding the `ToString()` method provides a way to quickly gain a snapshot of an object's current state. As you might guess, this can be helpful during the debugging process. To illustrate, let's override `System.Object.ToString()` to return a textual representation of a person's state (note we are using a new namespace named `System.Text`):

```
// Need to reference System.Text to access StringBuilder type.
using System;
using System.Text;

class Person
{
// Overriding System.Object.ToString().
public override string ToString()
{
StringBuilder sb = new StringBuilder();
sb.AppendFormat("[FirstName={0};", this.firstName);
sb.AppendFormat(" LastName={0};", this.lastName);
sb.AppendFormat(" SSN={0};", this.SSN);
sb.AppendFormat(" Age={0}]", this.age);
return sb.ToString();
}
...
}
```

How you format the string returned from `System.Object.ToString()` is largely a matter of personal choice. In this example, the name/value pairs have been contained within square brackets, with each pair separated by a semicolon (a common technique within the .NET base class libraries). Also notice that this example makes use of a new type, `System.Text.StringBuilder` (which is also a matter of personal choice). This type is

described in greater detail later in the chapter. The short answer, however, is that `StringBuilder` is a more efficient alternative to C# string concatenation.

Overriding System.Object.Equals()

Let's also override the behavior of `System.Object.Equals()` to work with *value-based semantics*. Recall that by default, `Equals()` returns true only if the two references being compared are pointing to the same object on the heap. In many cases, however, you don't necessary care if two references are pointing to the same object in memory, but you are more interested if the two objects have the same state data (`name`, `SSN`, and `age` in the case of a `Person`):

```
public override bool Equals(object o)
{
// Make sure the caller sent a valid
// Person object before proceeding.
if (o != null && o is Person)
{
// Now see if the incoming Person
// has the exact same information as
// the current object (this).
Person temp = (Person)o;
if (temp.firstName == this.firstName &&
temp.lastName == this.lastName &&
temp.SSN == this.SSN &&
temp.age == this.age)
return true;
}
return false;  // Not the same!
}
```

Here you are first verifying the caller did indeed pass in a `Person` object to the `Equals()` metho using the C# `is` keyword. After this point, you go about examining the values of the incoming parameter against the values of the current object's field data (note the use of the `this` keyword, which refers to the current object).

The prototype of `System.Object.Equals()` takes a single argument of type `object`. Thus, you are required to perform an explicit cast within the `Equals()` method to access the members of the `Person` type. If the name, SSN, and age of each are identical, you have two objects with the same state data and therefore return true. If any point of data is not identical, you return false. If you override `System.Object.ToString()` for a given class, you can take a very simple shortcut when overriding `System.Object.Equals()`. Given that the value returned from `ToString()` should take into account all of the member variables of the current class (and possible data declared in base classes), `Equals()` can simply compare the values of the string types:

```
public override bool Equals(object o)
{
if (o != null && o is Person)
{
if (this.ToString() == o.ToString())
return true;
```

```
else
return false;
}
return false;
}
```

Now, for the sake of argument, assume you have a type named `Car`, and attempt to pass in a `Car` instance to the `Person.Equals()` method as so:

```
// Cars are not people!
Car c = new Car();
Person p = new Person();
p.Equals(c);
```

Given your runtime check for a true-blue `Person` object (via the `is` operator) the `Equals()` method returns false. Now consider the following invocation:

```
// Oops!
Person p = new Person();
p.Equals(null);
```

This would also be safe, given your check for an incoming null reference.

Overriding System.Object.GetHashCode()

When a class overrides the `Equals()` method, best practices dictate that you should also override `System.Object.GetHashCode()`. If you fail to do so, you are issued a compilerwarning. The role of `GetHashCode()` is to return a numerical value that identifies an object based on its internal state data. Thus, if you have two `Person` objects that have an identical first name, last name, SSN, and age, you should obtain the same hash code.

There are many algorithms that can be used to create a hash code—some fancy, others not so fancy. As mentioned, an object's hash value will be based on its state data. As luck would have it, the `System.String` class has a very solid implementation of `GetHashCode()` that is based on the string's character data. Therefore, if you can identify a string field that should be unique among objects (such as the `Person`'s `SSN` field), you can simply call `GetHashCode()` on the field's string representation:

```
// Return a hash code based on the person's SSN.
public override int GetHashCode()
{
return SSN.GetHashCode();
}
```

If you cannot identify a single point of data in your class, but have overridden `ToString()`, you
can simply return the hash code of the string returned from your custom `ToString()` implementation:

```
// Return a hash code based our custom ToString().
public override int GetHashCode()
{
return ToString().GetHashCode();
}
```

3.10 Converting Between Value Types and Reference Types: Boxing and Unboxing Operations

Given that .NET defines two major categories of types (value based and reference based), you may occasionally need to represent a variable of one category as a variable of the other category. C# provides a very simple mechanism, termed *boxing*, to convert a value type to a reference type. Assume that you have created a variable of type `short`:

```
// Make a short value type.
short s = 25;
```

If, during the course of your application, you wish to represent this value type as a reference type, you would "box" the value as follows:

```
// Box the value into an object reference.
object objShort = s;
```

Boxing can be formally defined as the process of explicitly converting a value type into a corresponding reference type by storing the variable in a `System.Object`. When you box a value, the CLR allocates a new object on the heap and copies the value type's value (in this case, 25) into that instance. What is returned to you is a reference to the newly allocated object. Using this technique, .NET developers have no need to make use of a set of wrapper classes used to temporarily treat stack data as heap-allocated objects.

The opposiiten operation is also permitted through *unboxing*. Unboxing is the process of converting
the value held in the object reference back into a corresponding value type on the stack. The unboxing operation begins by verifying that the receiving data type is equivalent to the boxed type, and if so, it copies the value back into a local stack-based variable. For example, the following unboxing operation works successfully, given that the underlying type of the `objShort`is indeed a`short`(you'll examine the C# casting operator in detail in the next chapter, so hold tight for now):

```
// Unbox the reference back into a corresponding short.
short anotherShort = (short)objShort;
```

Again, it is mandatory that you unbox into an appropriate data type. Thus, the following
unboxing logic generates an `InvalidCastException` exception (more details on exception handling in Chapter 6):

```
// Illegal unboxing.
static void Main(string[] args)
{
...
try
{
// The type contained in the box is NOT a int, but a short!
int i = (int)objShort;
}
catch(InvalidCastException e)
{
Console.WriteLine("OOPS!\n{0} ", e.ToString());
}
```

```
}
```

Some Practical (Un)Boxing Examples

So, you may be thinking, when would you really need to manually box (or unbox) a data type? The previous example was purely illustrative in nature, as there was no good reason to box (and then unbox) the `short` data point. The truth of the matter is that you will seldom—if ever—need to manually box data types. Much of the time, the C# compiler automatically boxes variables when appropriate. For example, if you pass a value type into a method requiring an object parameter, boxing occurs behind the curtains.

```
class Program
{
static void Main(string[] args)
{
// Create an int (value type).
int myInt = 99;

// Because myInt is passed into a
// method prototyped to take an object,
// myInt is 'boxed' automatically.
UseThisObject(myInt);
Console.ReadLine();
}

static void UseThisObject(object o)
{
Console.WriteLine("Value of o is: {0}", o);
}
}
```

Automatic boxing also occurs when working with the types of the .NET base class libraries. For example, the `System.Collections` namespace (formally examined in Chapter 7) defines a class type named `ArrayList`. Like most collection types, `ArrayList` provides members that allow you to insert, obtain, and remove items:

```
public class System.Collections.ArrayList : object,
System.Collections.IList,
System.Collections.ICollection,
System.Collections.IEnumerable,
ICloneable
{
...
public virtual int Add(object value);
public virtual void Insert(int index, object value);
public virtual void Remove(object obj);
public virtual object this[int index] {get; set; }
}
```

As you can see, these members operate on generic `System.Object` types. Given that everything ultimately derives from this common base class, the following code is perfectly legal:

```
static void Main(string[] args)
{
...
ArrayList myInts = new ArrayList();
myInts.Add(88);
myInts.Add(3.33);
myInts.Add(false);
}
```

However, given your understanding of value types and reference types, you might wonder exactly what was placed into the `ArrayList` type. (References? Copies of references? Copies of structures?) Just like with the previous `UseThisObject()` method, it should be clear that each of the `System.Int32` data types were indeed boxed before being placed into the `ArrayList` type. To retrieve an item from the `ArrayList` type, you are required to unbox accordingly:

```
static void BoxAndUnboxInts()
{
// Box ints into ArrayList.
ArrayList myInts = new ArrayList();
myInts.Add(88);
myInts.Add(3.33);
myInts.Add(false);

// Unbox first item from ArrayList.
int firstItem = (int)myInts[0];
Console.WriteLine("First item is {0}", firstItem);
}
```

To be sure, boxing and unboxing types takes some processing time and, if used without restraint, could hurt the performance of your application. However, with this .NET technique, you are able to symmetrically operate on value-based and reference-based types.

3.11 Defining Program Constant

Now that you have seen how to declare class variables, let's see how to define data that should never be reassigned. C# offers the `const` keyword to define variables with a fixed, unalterable value. Once the value of a constant has been established, any attempt to alter it results in a compiler error. Unlike in C++, in C# the `const` keyword cannot be used to qualify parameters or return values, and is reserved for the creation of local or instance-level data. It is important to understand that the value assigned to a constant variable must be known at *compile time*, and therefore a constant member cannot be assigned to an object reference (whose value is computed at runtime). To illustrate the use of the `const` keyword, assume the following class type:

```
class ConstData
```

```
{
// The value assigned to a const must be known
// at compile time.
public const string BestNbaTeam = "Timberwolves";
public const double SimplePI = 3.14;
public const bool Truth = true;
public const bool Falsity = !Truth;
}
```

Notice that the value of each constant is known at the time of compilation. In fact, if you were to view these constants using `ildasm.exe`, you would find the value hard-coded directly into the assembly.

3.12 C# Iteration Constructs

All programming languages provide ways to repeat blocks of code until a terminating condition has been met. Regardless of which language you have used in the past, the C# iteration statements should not raise too many eyebrows and should require little explanation. C# provides the following
four iteration constructs:

- `for` loop
- `foreach/in` loop
- `while` loop
- `do/while` loop

Let's quickly examine each looping construct in turn.

The for Loop

When you need to iterate over a block of code a fixed number of times, the `for` statement is the construct of champions. In essence, you are able to specify how many times a block of code repeats itself, as well as the terminating condition. Without belaboring the point, here is a sample of the syntax:

```
// A basic for loop.
static void Main(string[] args)
{
// Note! 'i' is only visible within the scope of the for loop.
for(int i = 0; i < 10; i++)
{
Console.WriteLine("Number is: {0} ", i);
}
// 'i' is not visible here.
}
```

All of your old C, C++, and Java tricks still hold when building a C# `for` statement. You can create complex terminating conditions, build endless loops, and make use of the `goto`, `continue`, and `break` keywords. I'll assume that you will bend this iteration construct as you see fit. Consult the .NET Framework 2.0 SDK documentation if you require further details on the C# `for` keyword.

The foreach Loop

The C# `foreach` keyword allows you to iterate over all items within an array, without the need to test for the array's upper limit. Here are two examples using `foreach`, one to traverse an array of strings and the other to traverse an array of integers:

```
// Iterate array items using foreach.
static void Main(string[] args)
{
string[] books = {"Complex Algorithms",
"Do you Remember Classic COM?",
"C# and the .NET Platform"};
foreach (string s in books)
Console.WriteLine(s);

int[] myInts = { 10, 20, 30, 40 };
foreach (int i in myInts)
Console.WriteLine(i);
}
```

In addition to iterating over simple arrays, `foreach` is also able to iterate over system supplied or user-defined collections. I'll hold off on the details until Chapter 7, as this aspect of the `foreach` keyword entails an understanding of interface-based programming and the role of the `IEnumerator` and `IEnumerable` interfaces.

The while and do/while Looping Constructs

The `while` looping construct is useful should you wish to execute a block of statements until some terminating condition has been reached. Within the scope of a `while` loop, you will, of course, need to ensure this terminating event is indeed established; otherwise, you will be stuck in an endless loop. In the following example, the message "In while loop" will be continuously printed until the user terminates the loop by entering yes at the command prompt:

```
static void Main(string[] args)
{
string userIsDone = "no";

// Test on a lower class copy of the string.
while(userIsDone.ToLower() != "yes")
{
Console.Write("Are you done? [yes] [no]: ");
userIsDone = Console.ReadLine();
Console.WriteLine("In while loop");
}
}
```

Closely related to the `while` loop is the `do/while` statement. Like a simple `while` loop, `do/while` is used when you need to perform some action for an undetermined number of times. The difference is that `do/while` loops are guaranteed to execute the corresponding block of code at least once (in con- trast, it is possible that a simple `while` loop may never execute if the terminating condition is false from the onset).

```
static void Main(string[] args)
{
string userIsDone = "";
{
Console.WriteLine("In do/while loop");
Console.Write("Are you done? [yes] [no]: ");
userIsDone = Console.ReadLine();
}while(userIsDone.ToLower() != "yes");  // Note the semicolon!
}
```

3.13 C# Control flow Constructs

Now that you can iterate over a block of statements, the next related concept is how to control the flow of program execution. C# defines two simple constructs to alter the flow of your program, based on various contingencies:

- The `if/else` statement
- The `switch` statement

The if/else Statement

First up is our good friend the `if/else` statement. Unlike in C and C++, however, the `if/else` statement in C# operates only on Boolean expressions, not ad hoc values such as –1, 0. Given this, `if/else` statements typically involve the use of the C# operators shown in Table 3-6 in order to obtain a literal Boolean value.

Table 3-6. *C# Relational and Equality Operators*

C# Equality/Relational Operator	Example Usage	Meaning in Life
==	`if(age == 30)`	Returns `true` only if each expression is the same
!=	`if("Foo" != myStr)`	Returns `true` only if each expression is different
<	`if(bonus < 2000)`	Returns true if expression A is less than,
>	`if(bonus > 2000)`	greater than, less than or equal to,
<=	`if(bonus <= 2000)`	or greater than or equal to expression B
>=	`if(bonus >= 2000)`	

Again, C and C++ programmers need to be aware that the old tricks of testing a condition for a value “not equal to zero” will not work in C#. Let’s say you want to see if the string you are working with is longer than zero characters. You may be tempted to write \

```
// This is illegal, given that Length returns an int, not a bool.
string thoughtOfTheDay = "You CAN teach an old dog new tricks";
if(thoughtOfTheDay.Length)
{
```

```
...
}
```

If you wish to make use of the `String.Length` property to determine if you have an empty string, you need to modify your conditional expression as follows:

```
// Legal, as this resolves to either true or false.
if( 0 != thoughtOfTheDay.Length)
{
...
}
```

An `if` statement may be composed of complex expressions as well and can contain `else` statements to perform more-complex testing. The syntax is identical to C(++) and Java (and not too far removed from Visual Basic). To build complex expressions, C# offers an expected set of conditional operators, as shown in Table 3-7.

Table 3-7. *C# Conditional Operators*

Operator	Example	Meaning in Life
`&&`	`if((age == 30) && (name == "Fred"))`	Conditional AND operator
`||`	`if((age == 30) || (name == "Fred"))`	Conditional OR operator
`!`	`if(!myBool)`	Conditional NOT operator

The switch Statement

The other simple selection construct offered by C# is the `switch` statement. As in other C based languages, the `switch` statement allows you to handle program flow based on a predefined set of choices. For example, the following `Main()` logic prints a specific string message based on one of two possible selections (the `default` case handles an invalid selection):

```
// Switch on a numerical value.
static void Main(string[] args)
{
Console.WriteLine("1 [C#], 2 [VB]");
Console.Write("Please pick your language preference: ");

string langChoice = Console.ReadLine();
int n = int.Parse(langChoice);

switch (n)
{
case 1:
Console.WriteLine("Good choice, C# is a fine language.");
break;
case 2:
Console.WriteLine("VB .NET: OOP, multithreading, and more!");
break;
default:
```

```
Console.WriteLine("Well...good luck with that!");
break;
}
}
```

One nice feature of the C# `switch` statement is that you can evaluate string data in addition to numeric data. Here is an updated `switch` statement that does this very thing (notice we have no need to parse the user data into a numeric value with this approach):

```
static void Main(string[] args)
{
Console.WriteLine("C# or VB");
Console.Write("Please pick your language preference: ");

string langChoice = Console.ReadLine();
switch (langChoice)
{
case "C#":
Console.WriteLine("Good choice, C# is a fine language.");
break;
case "VB":
Console.WriteLine("VB .NET: OOP, multithreading and more!");
break;
default:
Console.WriteLine("Well...good luck with that!");
break;
}
}
```

3.14 The complete set of C

C# Shorthand	CLS Compliant?	System Type	Range	Meaning in Life
sbyte	No	System.SByte	–128 to 127	Signed 8-bit number
byte	Yes	System.Byte	0 to 255	Unsigned 8-bit number
short	Yes	System.Int16	–32,768 to 32,767	Signed 16-bit number
ushort	No	System.UInt16	0 to 65,535	Unsigned 16-bit number
int	Yes	System.Int32	–2,147,483,648 to 2,147,483,647	Signed 32-bit number
uint	No	System.UInt32	0 to 4,294,967,295	Unsigned 32-bit number
long	Yes	System.Int64	–9,223,372,036,854,775,808	Signed 64-bit number

to 9,223,372,036,854,775,807

`ulong`	No	`System.UInt64`	0 to 18,446,744,073,709,551,615	Unsigned 64-bit number
`char`	Yes	`System.Char`	U0000 to Uffff	A single 16-bit Unicode character
`float`	Yes	`System.Single`	1.5 10^{-45} to 3.4 10^{38}	32-bit floating point number
`double`	Yes	`System.Double`	5.0 10^{-324} to 1.7 10^{308}	64-bit floating point number
`bool`	Yes	`System.Boolean`	`true` or `false`	Represents truth or falsity
`decimal`	Yes	`System.Decimal`	10^{0} to 10^{28}	A 96-bit signed number
`string`	Yes	`System.String`	Limited by system memory	Represents a set of Unicode characters
`object`	Yes	`System.Object`	Any type can be stored in an object variable	The base class of all types in the .NET universe

3.15 Method Parameter Modifiers

Methods (static and instance level) tend to take parameters passed in by the caller. However, unlike some programming languages, C# provides a set of parameter modifiers that control how arguments are sent into (and possibly returned from) a given method, as shown in Table 3-5.

Table 3-5. *C# Parameter Modifiers*

Parameter Modifier	Meaning in Life
(none)	If a parameter is not marked with a parameter modifier, it is assumed to be *passed by value,* meaning the called method receives a copy of the original data.
`out`	Output parameters are assigned by the method being called (and therefore *passed by reference*). If the called method fails to assign output parameters, you are issued a compiler error.
`params`	This parameter modifier allows you to send in a variable number of identically typed arguments as a single logical parameter. A method can have only a single `params` modifier, and it must be the final parameter of

the method.

`ref` The value is initially assigned by the caller, and may be *optionally* reassigned by the called method (as the data is also passed by reference). No compiler error is generated if the called method fails to assign a `ref` parameter.

The Default Parameter-Passing Behavior

The default manner in which a parameter is sent into a function is *by value*. Simply put, if you do not mark an argument with a parameter-centric modifier, a copy of the variable is passed into the function:

```
// Arguments are passed by value by default.
public static int Add(int x, int y)
{
int ans = x + y;

// Caller will not see these changes
// as you are modifying a copy of the
// original data.
x = 10000;
y = 88888;
return ans;
}
```

Here, the incoming integer parameters will be passed by value. Therefore, if you change the values of the parameters within the scope of the member, the caller is blissfully unaware, given that you are changing the values of copies of the caller's integer data types:

```
static void Main(string[] args)
{
int x = 9, y = 10;
Console.WriteLine("Before call: X: {0}, Y: {1}", x, y);
Console.WriteLine("Answer is: {0}", Add(x, y));
Console.WriteLine("After call: X: {0}, Y: {1}", x, y);
}
```

As you would hope, the values of `x` and `y` remain identical before and after the call to `Add()`.

The out Modifier Next, we have the use of *output* parameters. Methods that have been defined to take output parameters are under obligation to assign them to an appropriate value before exiting the method in question (if you fail to ensure this, you will receive compiler errors). To illustrate, here is an alternative version of the `Add()` method that returns the sum of two integers using the C# `out` modifier (note the physical return value of this method is now `void`):

```
// Output parameters are allocated by the member.
public static void Add(int x, int y, out int ans)
{
ans = x + y;
}
```

Calling a method with output parameters also requires the use of the `out` modifier. Recall that local variables passed as output variables are not required to be assigned before use (if you do so, the original value is lost after the call), for example:

```
static void Main(string[] args)
{
// No need to assign local output variables.
int ans;
Add(90, 90, out ans);
Console.WriteLine("90 + 90 = {0} ", ans);
}
```

The previous example is intended to be illustrative in nature; you really have no reason to return the value of your summation using an output parameter. However, the C# `out` modifier does serve a very useful purpose: it allows the caller to obtain multiple return values from a single method invocation.

```
// Returning multiple output parameters.
public static void FillTheseValues(out int a, out string b, out bool c)
{
a = 9;
b = "Enjoy your string.";
c = true;
}
```

The caller would be able to invoke the following method:

```
static void Main(string[] args)
{
int i;
string str;
bool b;

FillTheseValues(out i, out str, out b);
Console.WriteLine("Int is: {0}", i);
Console.WriteLine("String is: {0}", str);
Console.WriteLine("Boolean is: {0}", b);
}
```

The ref Modifier

Now consider the use of the C# `ref` parameter modifier. Reference parameters are necessary when you wish to allow a method to operate on (and usually change the values of) various data points declared in the caller's scope (such as a sorting or swapping routine). Note the distinction between output and reference parameters:

- Output parameters do not need to be initialized before they passed to the method. The reason for this? The method must assign output parameters before exiting.
- Reference parameters *must* be initialized before they are passed to the method. The reason for this? You are passing a reference to an existing variable. If you don't assign it to an initial value, that would be the equivalent of operating on an unassigned local variable.

Let's check out the use of the `ref` keyword by way of a method that swaps two strings:

```
// Reference parameter.
public static void SwapStrings(ref string s1, ref string s2)
{
string tempStr = s1;
s1 = s2;
s2 = tempStr;
}
```

This method can be called as so:

```
static void Main(string[] args)
{
string s = "First string";
string s2 = "My other string";
Console.WriteLine("Before: {0}, {1} ", s, s2);
SwapStrings(ref s, ref s2);
Console.WriteLine("After: {0}, {1} ", s, s2);
}
```

Here, the caller has assigned an initial value to local string data (`s` and `s2`). Once the call to `SwapStrings()` returns, `s` now contains the value `"My other string"`, while `s2` reports the value `"First string"`.

The params Modifier

The final parameter modifier is the `params` modifier, which allows you to create a method that may be sent to a set of identically typed arguments *as a single logical parameter*. Yes, this can be confusing. To clear the air, assume a method that returns the average of any number of doubles:

```
// Return average of 'some number' of doubles.
static double CalculateAverage(params double[] values)
{
double sum = 0;
for (int i = 0; i < values.Length; i++)
sum += values[i];
return (sum / values.Length);
}
```

This method has been defined to take a parameter array of doubles. What this method is in fact saying is, "Send me *any number of doubles* and I'll compute the average." Given this, you can call `CalculateAverage()` in any of the following ways (if you did not make use of the `params` modifier in the definition of `CalculateAverage()`, the first invocation of this method would result in a compiler error):

```
static void Main(string[] args)
{
// Pass in a comma-delimited list of doubles...
double average;
average = CalculateAverage(4.0, 3.2, 5.7);
Console.WriteLine("Average of 4.0, 3.2, 5.7 is: {0}", average);
// ...or pass an array of doubles.
double[] data = { 4.0, 3.2, 5.7 };
```

```
average = CalculateAverage(data);
Console.WriteLine("Average of data is: {0}", average);
Console.ReadLine();
}
```

That wraps up our initial look at parameter modifiers. We'll revisit this topic later in the chapterwhen we examine the distinction between value types and reference types. Next up, let's check out the iteration and decision constructions of the C# programming language.

3.16 Array Manipulation in C#

Formally speaking, an *array* is a collection of data points, of the *same* defined data type, that are accessed using a numerical index. Arrays are references types and derive from a common base class named `System.Array`. By default, .NET arrays always have a lower bound of zero, although it is possible to create an array with an arbitrary lower bound using the static `System.Array.CreateInstance()` method.

C# arrays can be declared in a handful of ways. First of all, if you are creating an array whose values will be specified at a later time (perhaps due to yet-to-be-obtained user input), specify the size of the array using square brackets (`[]`) at the time of its allocation, for example:

```
// Assign a string array containing 3 elements {0 - 2}
string[] booksOnCOM;
booksOnCOM = new string[3];

// Initialize a 100 item string array, numbered {0 - 99}
string[] booksOnDotNet = new string[100];
```

Once you have declared an array, you can make use of the indexer syntax to fill each item with a value:

```
// Create, populate, and print an array of three strings.
string[] booksOnCOM;
booksOnCOM = new string[3];
booksOnCOM[0] = "Developer's Workshop to COM and ATL 3.0";
booksOnCOM[1] = "Inside COM";
booksOnCOM[2] = "Inside ATL";
foreach (string s in booksOnCOM)
Console.WriteLine(s);
```

As a shorthand notation, if you know an array's values at the time of declaration, you may specify these values within curly brackets. Note that in this case, the array size is optional (as it is calculated on the fly), as is the `new` keyword. Thus, the following declarations are identical:

```
// Shorthand array declaration (values must be known at time of declaration).
int[] n = new int[] { 20, 22, 23, 0 };
int[] n3 = { 20, 22, 23, 0 };
```

There is one final manner in which you can create an array type:

```
int[] n2 = new int[4] { 20, 22, 23, 0 }; // 4 elements, {0 - 3}
```

In this case, the numeric value specified represents the *number of elements* in the array, not the value of the upper bound. If there is a mismatch between the declared size and the number of initial- izers, you are issued a compile time error.

Regardless of how you declare an array, be aware that the elements in a .NET array are automatically set to their respective default values until you indicate otherwise. Thus, if you have an array of numerical types, each member is set to 0 (or 0.0 in the case of floating-point numbers), objects are set to null, and Boolean types are set to false.

Arrays As Parameters (and Return Values)

Once you have created an array, you are free to pass it as a parameter and receive it as a member return value. For example, the following `PrintArray()` method takes an incoming array of ints and prints each member to the console, while the `GetStringArray()` method= populates an array of strings and returns it to the caller:

```
static void PrintArray(int[] myInts)
{
for(int i = 0; i < myInts.Length; i++)
Console.WriteLine("Item {0} is {1}", i, myInts[i]);
}

static string[] GetStringArray()
{
string[] theStrings = { "Hello", "from", "GetStringArray" };
return theStrings;
}
```

These methods may be invoked from a `Main()` method as so:

```
static void Main(string[] args)
{
int[] ages = {20, 22, 23, 0} ;
PrintArray(ages);
string[] strs = GetStringArray();
foreach(string s in strs)
Console.WriteLine(s);
Console.ReadLine();
}
```

Working with Multidimensional Arrays

In addition to the single-dimension arrays you have seen thus far, C# also supports two varieties of multidimensional arrays. The first of these is termed a *rectangular array*, which is simply an array of multiple dimensions, where each row is of the same length. To declare and fill a multidimensional rectangular array, proceed as follows:

```
static void Main(string[] args)
{
...
// A rectangular MD array.
int[,] myMatrix;
myMatrix = new int[6,6];

// Populate (6 * 6) array.
for(int i = 0; i < 6; i++)
for(int j = 0; j < 6; j++)
```

```
myMatrix[i, j] = i * j;

// Print (6 * 6) array.
for(int i = 0; i < 6; i++)

{
for(int j = 0; j < 6; j++)
Console.Write(myMatrix[i, j] + "\t");
Console.WriteLine();
}
...
}
```

The second type of multidimensional array is termed a *jagged array*. As the name implies,
jagged arrays contain some number of inner arrays, each of which may have a unique upper limit, for example:

```
static void Main(string[] args)
{
...
// A jagged MD array (i.e., an array of arrays).
// Here we have an array of 5 different arrays.
int[][] myJagArray = new int[5][];

// Create the jagged array.
for (int i = 0; i < myJagArray.Length; i++)
myJagArray[i] = new int[i + 7];

// Print each row (remember, each element is defaulted to zero!)
for(int i = 0; i < 5; i++)
{
Console.Write("Length of row {0} is {1} :\t", i, myJagArray[i].Length);
for(int j = 0; j < myJagArray[i].Length; j++)
Console.Write(myJagArray[i][j] + " ");
Console.WriteLine();
}
}
```

Now that you understand how to build and populate C# arrays, let's turn our attention to the ultimate base class of any array: `System.Array`.

The System.Array Base Class

Every .NET array you create is automatically derived from `System.Array`. This class defines a number of helpful methods that make working with arrays much more palatable. Table 3-14 gives a rundown of some (but not all) of the more interesting members.

Table 3-14. *Select Members of* `System.Array`

Member	Meaning in Life

`BinarySearch()` This static method searches a (previously sorted) array for a given item.
If the array is composed of custom types you have created, the type in question must implement the `IComparer` interface (see Chapter 7) to engage in a binary search.

`Clear()` This static method sets a range of elements in the array to empty values (0 for value types; null for reference types).

`CopyTo()` This method is used to copy elements from the source array into the destination array.

`Length` This read-only property is used to determine the number of elements in an array.

`Rank` This property returns the number of dimensions of the current array.

`Reverse()` This static method reverses the contents of a one-dimensional array.

`Sort()` This method sorts a one-dimensional array of intrinsic types. If the elements in the array implement the `IComparer` interface, you can also sort your custom types (again, see Chapter 7).

Let's see some of these members in action. The following code makes use of the static `Reverse()` and `Clear()` methods (and the `Length` property) to pump out some information about an array of strings named `firstNames` to the console:

```
// Create some string arrays and exercise some System.Array members.
static void Main(string[] args)
{
// Array of strings.
string[] firstNames = { "Steve", "Dominic", "Swallow", "Baldy"} ;

// Print names as declared


Console.WriteLine("Here is the array:");
for(int i = 0; i < firstNames.Length; i++)
Console.Write("Name: {0}\t", firstNames[i]);
Console.WriteLine("\n");

// Reverse array and print.
Array.Reverse(firstNames);
Console.WriteLine("Here is the array once reversed:");
for(int i = 0; i < firstNames.Length; i++)
Console.Write("Name: {0}\t", firstNames[i]);
```

```
Console.WriteLine("\n");

// Clear out all but Baldy.
Console.WriteLine("Cleared out all but Baldy...");
Array.Clear(firstNames, 1, 3);
for(int i = 0; i < firstNames.Length; i++)
Console.Write("Name: {0}\t", firstNames[i]);
Console.ReadLine();
}
```

3.17 String Manipulation in C#

The C# `string` keyword is a shorthand notation of the `System.String` type, which provides a number of members you would expect from such a utility class. Table 3-12 lists some (but not all) of the interesting members.

Table 3-12. *Select Members of* `System.String`

Member	Meaning in Life
`Length`	This property returns the length of the current string.
`Contains()`	This method is used to determine if the current string object contains a specified string.
`Format()`	This static method is used to format a string literal using other primitives (i.e., numerical data and other strings) and the `{0}` notation examined earlier in this chapter.
`Insert()`	This method is used to receive a copy of the current string that contains newly inserted string data.
`PadLeft()` `PadRight()`	These methods return copies of the current string that has been padded with specific data.
`Remove()` `Replace()`	Use these methods to receive a copy of a string, with modifications (characters removed or replaced).
`Substring()`	This method returns a string that represents a substring of the current string.
`ToCharArray()`	This method returns a character array representing the current string.
`ToUpper()` `ToLower()`	These methods create a copy of a given string in uppercase or lowercase.

Basic String Operations

To illustrate some basic string operations, consider the following `Main()` method:

```
static void Main(string[] args)
{
Console.WriteLine("***** Fun with Strings *****");
string s = "Boy, this is taking a long time.";
```

```
Console.WriteLine("--> s contains 'oy'?: {0}", s.Contains("oy"));
Console.WriteLine("--> s contains 'Boy'?: {0}", s.Contains("Boy"));
Console.WriteLine(s.Replace('.', '!'));
Console.WriteLine(s.Insert(0, "Boy O' "));
Console.ReadLine();
}
```

You should be aware that although `string` is a reference type, the equality operators (`==` and `!=`) are defined to compare the *value* with the string objects, not the memory to which they refer. Therefore, the following comparison evaluates to true:

```
string s1 = "Hello ";
string s2 = "Hello ";
Console.WriteLine("s1 == s2: {0}", s1 == s2);
```

whereas this comparison evaluates to false:

```
string s1 = "Hello ";
string s2 = "World!";
Console.WriteLine("s1 == s2: {0}", s1 == s2);
```

When you wish to concatenate existing strings into a new string that is the sum of all its parts, C# provides the `+` operator as well as the static `String.Concat()` method. Given this, the following statements are functionally equivalent:

```
// Concatenation of strings.
string newString = s + s1 + s2;
Console.WriteLine("s + s1 + s2 = {0}", newString);
Console.WriteLine("string.Concat(s, s1, s2) = {0}", string.Concat(s, s1, s2));
```

Another helpful feature of the `string` type is the ability to iterate over each individual character using an arraylike syntax. Formally speaking, objects that support arraylike access to their contents make use of an *indexer method*. You'll learn how to build indexers in Chapter 9; however, to illustrate the concept, the following code prints each character of the `s1` string object to the console:

```
// System.String also defines an indexer to access each
// character in the string.
for (int k = 0; k < s1.Length; k++)
Console.WriteLine("Char {0} is {1}", k, s1[k]);
```

As an alternative to interacting with the type's indexer, the string class can also be used within the C# `foreach` construct. Given that `System.String` is maintaining an array of individual `System.Char` types, the following code also prints each character of `s1` to the console:

```
foreach(char c in s1)
Console.WriteLine(c);
```

Escape Characters

Like in other C-based languages, in C# string literals may contain various escape characters, which qualify how the character data should be printed to the output stream. Each escape character begins with a backslash, followed by a specific token. In case you are a bit rusty on the meanings behind these escape characters, Table 3-13 lists the more common options.

Table 3-13. *String Literal Escape Characters*

Character	Meaning in Life
\'	Inserts a single quote into a string literal.
\"	Inserts a double quote into a string literal.
\\	Inserts a backslash into a string literal. This can be quite helpful when defining file paths.
\a	Triggers a system alert (beep). For console applications, this can be an audio clue to the user.
\n	Inserts a new line (on Win32 platforms).
\r	Inserts a carriage return.
\t	Inserts a horizontal tab into the string literal.

For example, to print a string that contains a tab between each word, you can make use of the \t escape character:

```
// Literal strings may contain any number of escape characters.
string s3 = "Hello\tThere\tAgain";
Console.WriteLine(s3);
```

As another example, assume you wish to create a string literal that contains quotation marks, another that defines a directory path, and a final string literal that inserts three blank lines after printing the character data. To do so without compiler errors, you would need to make use of the \",
\\, and \n escape characters:

```
Console.WriteLine("Everyone loves \"Hello World\"");
Console.WriteLine("C:\\MyApp\\bin\\debug");
Console.WriteLine("All finished.\n\n\n");
```

Working with C# Verbatim Strings

C# introduces the @-prefixed string literal notation termed a *verbatim string*. Using verbatim strings, you disable the processing of a literal's escape characters. This can be most useful = when working with strings representing directory and network paths. Therefore, rather than making use of \\ escape characters, you can simply write the following: `// The following string is printed verbatim`

```
// thus, all escape characters are displayed.
Console.WriteLine(@"C:\MyApp\bin\debug");
```

Also note that verbatim strings can be used to preserve white space for strings that flow over multiple lines:

```
// White space is preserved with verbatim strings.
string myLongString = @"This is a very
very
very
long string";
Console.WriteLine(myLongString);
```

You can also insert a double quote into a literal string by doubling the " token, for example:

The Role of System.Text.StringBuilder

While the `string` type is perfect when you wish to represent basic string variables (first name, SSN, etc.), it can be inefficient if you are building a program that makes heavy use of textual = data. The reason has to do with a very important fact regarding .NET strings: the value of a string cannot be modified once established. C# strings are immutable.

On the surface, this sounds like a flat-out lie, given that we are always assigning new values to string variables. However, if you examine the methods of `System.String`, you notice that the methods that *seem* to internally modify a string in fact return a modified *copy* of the original string. For example, when you call `ToUpper()` on a string object, you are not= modifying the underlying buffer of an existing string object, but receive a new string object in uppercase form:

```
static void Main(string[] args)
{
...
// Make changes to strFixed? Nope!
System.String strFixed = "This is how I began life";
Console.WriteLine(strFixed);
string upperVersion = strFixed.ToUpper();
Console.WriteLine(strFixed);
Console.WriteLine("{0}\n\n", upperVersion);
...
}
```

In a similar vein, when you assign an existing string object to a new value, you have actually
allocated a *new* string in the process (the original string object will eventually be garbage collected). A similar process occurs with string concatenation.

To help reduce the amount of string copying, the `System.Text` namespace defines a class named `StringBuilder` (first seen during our examination of `System.Object` earlier in this chapter). Unlike `System.String`, `StringBuilder` provides you direct access to the underlying buffer. Like `System.String`, `StringBuilder` provides numerous members that allow you to append, format, insert, and remove data from the object (consult the .NET Framework 2.0 SDK documentation for full details).

When you create a `StringBuilder` object, you may specify (via a constructor argument) the initial number of characters the object can contain. If you do not do so, the default capacity of a `StringBuilder` is 16. In either case, if you add more character data to a `StringBuilder` than it = is able to hold, the buffer is resized on the fly. Here is an example of working with this class type:

```
using System;
using System.Text;    // StringBuilder lives here.

class StringApp
{
static void Main(string[] args)
{
StringBuilder myBuffer = new StringBuilder("My string data");
Console.WriteLine("Capacity of this StringBuilder: {0}",
```

```
myBuffer.Capacity);
myBuffer.Append(" contains some numerical data: ");
myBuffer.AppendFormat("{0}, {1}.", 44, 99);
Console.WriteLine("Capacity of this StringBuilder: {0}",
myBuffer.Capacity);
Console.WriteLine(myBuffer);
}
}
```

Now, do understand that in many cases, `System.String` will be your textual object of choice. For most applications, the overhead associated with returning modified copies of character data will be negligible. However, if you are building a text-intensive application (such as a word processor program), you will most likely find that using `System.Text.StringBuilder` improves performance.

3.18 Working with .NET Enumerations

In addition to structures, *enumerations* (or simply *enums*) are the other member of the .NET value type category. When you build a program, it is often convenient to create a set of symbolic names for underlying numerical values. For example, if you are creating an employee payroll system, you may wish to use the constants `Manager`, `Grunt`, `Contractor`, and `VP` rather than simple numerical values such as `{0, 1, 2, 3}`. C# supports the notion of = custom enumerations for this very reason. For example, here is the `EmpType` enumeration:

```
// A custom enumeration.
enum EmpType
{
Manager,     // = 0
Grunt,      // = 1
Contractor,   // = 2
VP         // = 3
}
```

The `EmpType` enumeration defines four named constants corresponding to specific numerical values. In C#, the numbering scheme sets the first element to zero (0) by default, followed by an *n + 1* progression. You are free to change this behavior as you see fit:

```
// Begin numbering at 102.
enum EmpType
{
Manager = 102,
Grunt,       // = 103
Contractor,    // = 104
VP          // = 105
}
```

Enumerations do not necessarily need to follow a sequential order. If (for some reason) it made good sense to establish your `EmpType` as follows, the compiler continues to be happy:

```
// Elements of an enumeration need not be sequential!
enum EmpType
{
Manager = 10,
Grunt = 1,
Contractor = 100,
VP = 9
}
```

Under the hood, the storage type used for each item in an enumeration maps to a `System.Int32` by default. You are also free to change this to your liking. For example, if you want to set the underlying storage value of `EmpType` to be a `byte` rather than an `int`, you would write the following:

```
// This time, EmpType maps to an underlying byte.
enum EmpType : byte
{
Manager = 10,
Grunt = 1,
Contractor = 100,
VP = 9
}
```

Note C# enumerations can be defined in a similar manner for any of the numerical types (`byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long`, or `ulong`). This can be helpful if you are programming for low-memory devices such as Pocket PCs or .NET-enabled cellular phones.

Once you have established the range and storage type of your enumeration, you can use them in place of so-called magic numbers. Assume you have a class defining a static function, taking `EmpType` as the sole parameter:

```
static void AskForBonus(EmpType e)
{
switch(e)
{
case EmpType.Contractor:
Console.WriteLine("You already get enough cash...");
break;
case EmpType.Grunt:
Console.WriteLine("You have got to be kidding...");
break;
case EmpType.Manager:
Console.WriteLine("How about stock options instead?");
break;
case EmpType.VP:
Console.WriteLine("VERY GOOD, Sir!");
break;
default: break;
}
}
```

This method can be invoked as so:

```
static void Main(string[] args)
{
// Make a contractor type.
EmpType fred;
fred = EmpType.Contractor;
AskForBonus(fred);
}
```

The System.Enum Base Class

The interesting thing about .NET enumerations is that they implicitly derive from `System.Enum`. This base class defines a number of methods that allow you to interrogate and transform a given enumeration. Table 3-9 documents some items of interest, all of which are static.

Table 3-9. *Select Static Members of* `System.Enum`

Member	Meaning in Life
`Format()`	Converts a value of a specified enumerated type to its equivalent string representation according to the specified format
`GetName()` `GetNames()`	Retrieves a name (or an array containing all names) for the constant in the specified enumeration that has the specified value
`GetUnderlyingType()`	Returns the underlying data type used to hold the values for a given enumeration
`GetValues()`	Retrieves an array of the values of the constants in a specified enumeration
`IsDefined()`	Returns an indication of whether a constant with a specified value exists in a specified enumeration
`Parse()`	Converts the string representation of the name or numeric value of one or more enumerated constants to an equivalent enumerated object

You can make use of the static `Enum.Format()` method and the same exact string formatting flags examined earlier in the chapter during our examination of `System.Console`. For example, you may extract the string name (by specifying `G`), the hexadecimal value (`X`), or numeric value (`D`, `F`, etc.) of a given enum. `System.Enum` also defines a static method named `GetValues()`. This method returns an instance of `System.Array` (examined later in this chapter), with each item in the array corresponding to name/value npairs of the specified enumeration. To illustrate these points, ponder the following:

```
static void Main(string[] args)
```

```
{
// Print information for the EmpType enumeration.
Array obj = Enum.GetValues(typeof(EmpType));
Console.WriteLine("This enum has {0} members.", obj.Length);

foreach(EmpType e in obj)
{
Console.Write("String name: {0},", e.ToString());
Console.Write(" int: ({0}),", Enum.Format(typeof(EmpType), e, "D"));
Console.Write(" hex: ({0})\n", Enum.Format(typeof(EmpType), e, "X"));
}
}
```

As you can guess, this code block prints out the name/value pairs (in decimal and hexadecimal)

Next, let's explore the `IsDefined()` method. This property allows you to determine if a given string name is a member of the current enumeration. For example, assume you wish to know if the value `SalesPerson` is part of the `EmpType` enumeration. To do so, you must send it the type information of the enumeration (which can be done via the C# `typeof` operator) and the string name of the value you wish to query (type information will be examined in much greater detail in Chapter 12):

```
static void Main(string[] args)
{
...
// Does EmpType have a SalesPerson value?
if(Enum.IsDefined(typeof(EmpType), "SalesPerson"))
Console.WriteLine("Yep, we have sales people.");
else
Console.WriteLine("No, we have no profits...");
}
```

It is also possible to generate an enumeration set to the correct value from a string literal via
the static `Enum.Parse()` method. Given that `Parse()` returns a generic `System.Object`, you will need to cast the return value into the correct enum type:

```
// Prints: "Sally is a Manager"
EmpType sally = (EmpType)Enum.Parse(typeof(EmpType), "Manager");
Console.WriteLine("Sally is a {0}", sally.ToString());
```

Last but not least, it is worth pointing out that C# enumerations support the use of various operators, which test against the assigned values, for example:

```
static void Main(string[] args)
{
...
// Which of these two EmpType variables has the greatest numerical value?
EmpType Joe = EmpType.VP;
EmpType Fran = EmpType.Grunt;

if(Joe < Fran)
```

```
Console.WriteLine("Joe's value is less than Fran's value.");
else
Console.WriteLine("Fran's value is less than Joe's value.");
}
```

3.19 Defining Structures in C#

The numerical types of .NET support `MaxValue` and `MinValue` properties that provide information regarding the range a given type can store. Assume you have created some variables of type `System.UInt16` (an `unsigned short`) and exercised it as follows:

```
static void Main(string[] args)
{
System.UInt16 myUInt16 = 30000;
Console.WriteLine("Max for an UInt16 is: {0} ", UInt16.MaxValue);
Console.WriteLine("Min for an UInt16 is: {0} ", UInt16.MinValue);
Console.WriteLine("Value is: {0} ", myUInt16);
Console.WriteLine("I am a: {0} ", myUInt16.GetType());

// Now in System.UInt16 shorthand (e.g., a ushort).
ushort myOtherUInt16 = 12000;
Console.WriteLine("Max for an UInt16 is: {0} ", ushort.MaxValue);
Console.WriteLine("Min for an UInt16 is: {0} ", ushort.MinValue);
Console.WriteLine("Value is: {0} ", myOtherUInt16);
Console.WriteLine("I am a: {0} ", myOtherUInt16.GetType());
Console.ReadLine();
}
```

In addition to the `MinValue/MaxValue` properties, a given system type may define further useful members. For example, the `System.Double` type allows you to obtain the values for Epsilon and infinity values:

```
Console.WriteLine("-> double.Epsilon: {0}", double.Epsilon);
Console.WriteLine("-> double.PositiveInfinity: {0}", double.PositiveInfinity);
Console.WriteLine("-> double.NegativeInfinity: {0}", double.NegativeInfinity);
Console.WriteLine("-> double.MaxValue: {0}", double.MaxValue);
Console.WriteLine("-> double.MinValue: {0}",double.MinValue);
```

Next, consider the `System.Boolean` data type. Unlike C(++), the only valid assignment a C# `bool` can take is from the set `{true | false}`. You cannot assign makeshift values (e.g., –1, 0, 1) to a C# `bool`, which (to most programmers) is a welcome change. Given this point, it should be clear that `System.Boolean` does not support a `MinValue/MaxValue` property set, but rather `TrueString/FalseString`:

```
// No more ad hoc Boolean types in C#!
bool b = 0;      // Illegal!
bool b2 = -1;     // Also illegal!
bool b3 = true;   // No problem.
bool b4 = false;  // No problem.
Console.WriteLine("-> bool.FalseString: {0}", bool.FalseString);
```

```
Console.WriteLine("-> bool.TrueString: {0}", bool.TrueString);
```

C# textual data is represented by the intrinsic C# string and `char` data types. All .NET-aware languages map textual data to the same underlying types (`System.String` and `System.Char`), both of which are Unicode under the hood.
The `System.Char` type provides you with a great deal of functionality beyond the ability to hold a single point of character data (which must be placed between single quotes). Using the static methods of `System.Char`, you are able to determine if a given character is numerical, alphabetical, a point of punctuation, or whatnot. To illustrate, check out the following:

```
static void Main(string[] args)
{
...
// Test the truth of the following statements...
Console.WriteLine("-> char.IsDigit('K'): {0}", char.IsDigit('K'));
Console.WriteLine("-> char.IsDigit('9'): {0}", char.IsDigit('9'));
Console.WriteLine("-> char.IsLetter('10', 1): {0}", char.IsLetter("10", 1));
Console.WriteLine("-> char.IsLetter('p'): {0}", char.IsLetter('p'));
Console.WriteLine("-> char.IsWhiteSpace('Hello There', 5): {0}",
char.IsWhiteSpace("Hello There", 5));
Console.WriteLine("-> char.IsWhiteSpace('Hello There', 6): {0}",
char.IsWhiteSpace("Hello There", 6));
Console.WriteLine("-> char.IsLetterOrDigit('?'): {0}",
char.IsLetterOrDigit('?'));
Console.WriteLine("-> char.IsPunctuation('!'): {0}",
char.IsPunctuation('!'));
Console.WriteLine("-> char.IsPunctuation('>'): {0}",
char.IsPunctuation('>'));
Console.WriteLine("-> char.IsPunctuation(','): {0}",
char.IsPunctuation(','));
...
}
```

As you can see, each of these static members of `System.Char` has two calling conventions: a single character or a string with a numerical index that specified the position of the character to test.

3.20 Defining Custom Namespaces

Up to this point, you have been building small test programs leveraging existing namespaces in the .NET universe (`System` in particular). When you build your own custom = applications, it can be very helpful to group your related types into custom namespaces. In C#, this is accomplished using the `namespace` keyword.
Assume you are developing a collection of geometric classes named `Square`, `Circle`, and `Hexagon`. Given their similarities, you would like to group them all together into a common custom namespace. You have two basic approaches. First, you may choose to define each class within a single file (`shapes-lib.cs`) as follows:

```
// shapeslib.cs
using System;

namespace MyShapes
{
// Circle class.
class Circle{  /* Interesting methods... */ }
// Hexagon class.
class Hexagon{ /* More interesting methods... */ }
// Square class.
class Square{  /* Even more interesting methods... */ }
```

Notice how the `MyShapes` namespace acts as the conceptual "container" of these types. Alternatively, you can split a single namespace into multiple C# files. To do so, simply wrap the given class definitions in the same namespace:

```
// circle.cs
using System;

namespace MyShapes
{
// Circle class.
class Circle{ }
}

// hexagon.cs
using System;

namespace MyShapes
{
// Hexagon class.
class Hexagon{ }
}

// square.cs
using System;

namespace MyShapes
{
// Square class.
class Square{ }
}
```

As you already know, when another namespace wishes to use objects within a distinct namespace, the `using` keyword can be used as follows:

```
// Make use of types defined the MyShape namespace.
using System;
using MyShapes;
```

```
namespace MyApp
{
class ShapeTester
{
static void Main(string[] args)
{
Hexagon h = new Hexagon();
Circle c = new Circle();
Square s = new Square();
}
}
}
```

Recommended questions

1. What are the key features C#?
2. What are the basic building blocks of .NET framework Visual Studio and explain.
3. Explain the csingle file and concept of .NET binaries.Differentiate between single file and multifile assemblies.
4. What is the role of .NET Type Metadata and Assembly Manifest
5. What are the CTS class characteristics

UNIT – 4

OOP with C#

4.1 Defining of the C# Class,
4.2 Definition the "Default Public Interface" of a Type
4.3 Recapping the Pillars of OOP,
4.4 The First Pillars: C#'s Encapsulation Services, Pseudo-Encapsulation:Creating Read-Only Fields
4.5The Second Pillar:C#'s Inheritance Supports
4.8keeping Family Secrets: The " Protected" Keyword, Nested Type Definitions,
4.9The Third Pillar: C #'s Polymorphic Support,
4.8 Casting Between.

4.1 Defition of the C# Class

If you have been "doing objects" in another programming language, you are no doubt aware of the role of class definitions. Formally, a class is nothing more than a custom user-defined type (UDT) that is composed of field data (sometimes termed *member variables*) and functions (often called *methods* in OO speak) that act on this data. The set of field data collectively represents the "state" of a class instance.

The power of object-oriented languages is that by grouping data and functionality in a single UDT, you are able to model your software types after real-world entities. For example, assume you are interested in modeling a generic employee for a payroll system. At minimum, you may wish to build a class that maintains the name, current pay, and= employee ID for each worker. In addition, the `Employee` class defines one method, named `GiveBonus()`, which increases an individual's current pay by some amount, and another, = named `DisplayStats()`, which prints out the state data for this individual.

```
// The initial Employee class definition.
namespace Employees
{
public class Employee
{
// Field data.
private string fullName;
private int empID;
private float currPay;

// Constructors.
public Employee(){ }
public Employee(string fullName, int empID, float currPay)
{
this.fullName = fullName;
this.empID = empID;
this.currPay = currPay;
}

// Bump the pay for this employee.
public void GiveBonus(float amount)
{ currPay += amount; }

// Show current state of this object.
public void DisplayStats()
{
Console.WriteLine("Name: {0} ", fullName);
Console.WriteLine("Pay: {0} ", currPay);
Console.WriteLine("ID: {0} ", empID);
}
```

```
}
}
public class Employee
{
...
public Employee(){ }
...
}
```

Like C++ and Java, if you choose to define custom constructors in a class definition, the default constructor is *silently removed*. Therefore, if you wish to allow the object user to create an instance of your class as follows:

```
static void Main(string[] args)
{
// Calls the default constructor.
Employee e = new Employee();
}
```

you must explicitly redefine the default constructor for your class (as we have done here). If you do not, you will receive a compiler error when creating an instance of your class type = using the default constructor. In any case, the following `Main()` method creates a few = `Employee` objects using our cus-
tom three-argument constructor:

```
// Make some Employee objects.
static void Main(string[] args)
{
Employee e = new Employee("Joe", 80, 30000);
Employee e2;
e2 = new Employee("Beth", 81, 50000);
Console.ReadLine();
}
```

4.2 Defining the Public Interface of a Class

Once you have established a class's internal state data and constructor set, your next step is to flesh out the details of the *public interface* to the class. The term refers to the set of member that are directly accessible from an object variable via the dot operator. From the class builder's point of view, the public interface is any item declared in a class using the `public` keyword. Beyond field data and constructors, the public interface of a class may be pop-ulated by numerous members, including the following:

- *Methods*: Named units of work that model some behavior of a class
- *Properties*: Traditional accessor and mutator functions in disguise
- *Constants/Read-only fields*: Field data that cannot be changed after assignment

Given that our `Employee` currently defines two public methods (`GiveBonus()` and `DisplayStats()`),
we are able to interact with the public interface as follows:

```
// Interact with the public interface of the Employee class type.
static void Main(string[] args)
```

```
{
Console.WriteLine("***** The Employee Type at Work *****\n");
Employee e = new Employee("Joe", 80, 30000);
e.GiveBonus(200);
e.DisplayStats();

Employee e2;
e2 = new Employee("Beth", 81, 50000);
e2.GiveBonus(1000);
e2.DisplayStats();
Console.ReadLine();
}
```

4.3 Recapping the Pillars of OOP

All object-oriented languages contend with three core principles of object-oriented programming, often called the famed "pillars of OOP."

- *Encapsulation*: How does this language hide an object's internal implementation?
- *Inheritance*: How does this language promote code reuse?
- *Polymorphism*: How does this language let you treat related objects in a similar way?

Before digging into the syntactic details of each pillar, it is important you understand the = basic role of each. Therefore, here is a brisk, high-level rundown, just to clear off any = cobwebs you may have acquired between project deadlines.

Encapsulation

The first pillar of OOP is called *encapsulation*. This trait boils down to the language's ability to hide unnecessary implementation details from the object user. For example, assume you are using a class named `DatabaseReader` that has two methods named `Open()` and `Close()`:

```
// DatabaseReader encapsulates the details of database manipulation.
DatabaseReader dbObj = new DatabaseReader();

dbObj.Open(@"C:\Employees.mdf");
// Do something with database...
dbObj.Close();
```

The fictitious `DatabaseReader` class has encapsulated the inner details of locating, loading, manipulating, and closing the data file. Object users love encapsulation, as this pillar of OOP keeps programming tasks simpler. There is no need to worry about the numerous lines of code that are working behind the scenes to carry out the work of the `DatabaseReader` class. All you do is create an instance and send the appropriate messages (e.g., "open the file named Employees.mdf located on my C drive").

Another aspect of encapsulation is the notion of data protection. Ideally, an object's state data should be defined as *private* rather than *public* (as was the case in previous chapters). In this way, the outside world must "ask politely" in order to change or obtain the underlying value.

Inheritance

The next pillar of OOP, inheritance, boils down to the language's ability to allow you to build new class definitions based on existing class definitions. In essence, inheritance allows you to extend the behavior of a base (or *parent*) class by enabling a subclass to inherit core functionality (also called a *derived class* or *child class*).
For example, if you are modeling an automobile, you might wish to express the idea that
a car "has-a" radio. It would be illogical to attempt to derive the `Car` class from a `Radio`, or vice versa.
(A `Car` "is-a" `Radio`? I think not!) Rather, you have two independent classes working together, where the containing class creates and exposes the contained class's functionality:

```
public class Radio
{
public void Power(bool turnOn)
{ Console.WriteLine("Radio on: {0}", turnOn);}
}

public class Car
{
// Car "has-a" Radio.
private Radio myRadio = new Radio();

public void TurnOnRadio(bool onOff)
{
// Delegate to inner object.
myRadio.Power(onOff);
}
}
```

Here, the containing type (`Car`) is responsible for creating the contained object (`Radio`). If the `Car` wishes to make the `Radio`'s behavior accessible from a `Car` instance, it must extend its own public interface with some set of functions that operate on the contained type. Notice that the object user has no clue that the `Car` class is making use of an inner `Radio` object:

```
static void Main(string[] args)
{
// Call is forward to Radio internally.
Car viper = new Car();
viper.TurnOnRadio(true);
}
```

Polymorphism

The final pillar of OOP is *polymorphism*. This trait captures a language's ability to treat related objects the same way. This tenent of an object-oriented language allows a base class to define a set of members (formally termed the *polymorphic interface*) to all descendents. A class type's polymorphic interface is constructed using any number of *virtual* or *abstract* members. In a nutshell, a virtual member *may* be changed (or more formally speaking, *overridden*) by a derived class, whereas an abstract method

must be overriden by a derived type. When derived types override the members defined by a base class, they are essentially redefining how they respond to the same request.

4.4 The First Pillar: C#'s Encapsulation Services

The concept of encapsulation revolves around the notion that an object's field data should not be directly accessible from the public interface. Rather, if an object user wishes to alter the state of an object, it does so indirectly using accessor (get) and mutator (set) methods. In C#, encapsulation is enforced at the syntactic level using the `public`, `private`, `protected`, and `protected internal` keywords, as described in Chapter 3. To illustrate the need for encapsulation, assume you have created the following class definition:

```
// A class with a single public field.
public class Book
{
public int numberOfPages;
}
```

The problem with public field data is that the items have no ability to "understand" whether
the current value to which they are assigned is valid with regard to the current business rules of the system. As you know, the upper range of a C# `int` is quite large (2,147,483,647). Therefore, the compiler allows the following assignment:

```
// Humm...
static void Main(string[] args)
{
Book miniNovel = new Book();
miniNovel.numberOfPages = 30000000;
}
```

Although you do not overflow the boundaries of an integer data type, it should be clear that a mini-novel with a page count of 30,000,000 pages is a bit unreasonable in the real world. As you can see, public fields do not provide a way to enforce data validation rules. If your system has a business rule that states a mini-novel must be between 1 and 200 pages, you are at a loss to enforce this programmatically. Because of this, public fields typically have no place in a production-level class definition (public read-only fields being the exception). Encapsulation provides a way to preserve the integrity of state data. Rather than defining public fields (which can easily foster data corruption), you should get in the habit of defining *private data fields*, which are indirectly manipulated by the caller using one of two main techniques:

- Define a pair of traditional accessor and mutator methods.
- Define a named property.

Whichever technique you choose, the point is that a well-encapsulated class should hide its
raw data and the details of how it operates from the prying eyes of the outside world. This is often termed *black box programming*. The beauty of this approach is that a class author is free to change how a given method is implemented under the hood,

without breaking any existing code making use of it (provided that the signature of the method remains constant).

Enforcing Encapsulation Using Traditional Accessors and Mutators

Let's return to the existing `Employee` class. If you want the outside world to interact with your private `fullName` data field, tradition dictates defining an *accessor* (get method) and *mutator* (set method). For example:

```
// Traditional accessor and mutator for a point of private data.
public class Employee
{
private string fullName;
...
// Accessor.
public string GetFullName() { return fullName; }

// Mutator.
public void SetFullName(string n)
{
// Remove any illegal characters (!, @, #, $, %),
// check maximum length (or case rules) before making assignment.
fullName = n;
}
}
```

Understand, of course, that the compiler could not care less what you call your accessor and
mutator methods. Given the fact that `GetFullName()` and `SetFullName()` encapsulate a private string named `fullName`, this choice of method names seems to fit the bill. The calling logic is as follows:

```
// Accessor/mutator usage.
static void Main(string[] args)
{
Employee p = new Employee();
p.SetFullName("Fred Flintstone");
Console.WriteLine("Employee is named: {0}", p.GetFullName());
Console.ReadLine();
}
```

Another Form of Encapsulation: Class Properties

In contrast to traditional accessor and mutator methods, .NET languages prefer to enforce encapsulation using *properties*, which simulate publicly accessible points of data. Rather than requiring the user to call two different methods to get and set the state data, the user is able to call what appears to be a public field. To illustrate, assume you have provided a property named `ID` that wraps the internal `empID` member variable of the `Employee` type. The calling syntax would look like this:

```
// Setting / getting a person's ID through property syntax.
static void Main(string[] args)
```

```
{
Employee p = new Employee();

// Set the value.
p.ID = 81;

// Get the value.
Console.WriteLine("Person ID is: {0} ", p.ID);
Console.ReadLine();
}
```

Type properties always map to "real" accessor and mutator methods under the hood. Therefore, as a class designer you are able to perform any internal logic necessary before making the value assignment (e.g., uppercase the value, scrub the value for illegal characters, check the bounds of a numerical value, and so on). Here is the C# syntax behind the `ID` property, another property (`Pay`) that encapsulates the `currPay` field, and a final property (`Name`) to encapsulate the `fullName` data point.

```
// Encapsulation with properties.
public class Employee
{
...
private int empID;
private float currPay;
private string fullName;

// Property for empID.
public int ID
{
get { return empID;}
set
{
// You are still free to investigate (and possibly transform)
// the incoming value before making an assignment.
empID = value;
}
}

// Property for fullName.
public string Name
{
get {return fullName;}
set {fullName = value;}
}

// Property for currPay.
public float Pay
{
get {return currPay;}
```

```
set {currPay = value;}
}
}
```

A C# property is composed using a get block (accessor) and set block (mutator). The C# "value" token represents the right-hand side of the assignment. The underlying data type of the value token depends on which sort of data it represents. In this example, the `ID` property is operating on a `int` data type, which, as you recall, maps to a `System.Int32`:

```
// 81 is a System.Int32, so "value" is a System.Int32.
Employee e = new Employee();
e.ID = 81;
```

To prove the point, assume you have updated the `ID` property's set logic as follows:

```
// Property for the empID.
public int ID
{
get { return empID;}
set
{
Console.WriteLine("value is an instance of: {0} ", value.GetType());
Console.WriteLine("value's value: {0} ", value);

empID = value;
}
}
```

Read-Only and Write-Only Properties

When creating class types, you may wish to configure a read-only property. To do so, simply build a property without a corresponding set block. Likewise, if you wish to have a write-only property, omit the get block. We have no need to do so for this example; however, here is how the `SocialSecurityNumber` property could be retrofitted as read-only:

```
public class Employee
{
...
// Now as a read-only property.
public string SocialSecurityNumber {  get {  return empSSN; }  }
}
```

Given this adjustment, the only manner in which an employee's US Social Security number can be set is through a constructor argument.

4.5 The Second Pillar: C#'s Inheritance Support

Now that you have seen various techniques that allow you to create a single well-encapsulated class, it is time to turn your attention to building a family of related classes. As mentioned, inheri- tance is the aspect of OOP that facilitates code reuse. Inheritance comes in two flavors: classical inheritance (the "is-a" relationship) and the

containment/delegation model (the "has-a" relationship). Let's begin by examining the classical "is-a" model.
When you establish "is-a" relationships between classes, you are building a dependency
between types. The basic idea behind classical inheritance is that new classes may leverage (and possibily extend) the functionality of other classes. To illustrate, assume that you wish to leverage the functionality of the `Employee` class to create two new classes.
For our example, we will assume that the `Manager` class extends `Employee` by recording the number of stock options, while the `SalesPerson` class maintains the number of sales. In C#, extending a class is accomplished using the colon operator (:) on the class definition. This being said, here are the derived class types:

```
// Add two new subclasses to the Employees namespace.
namespace Employees
{
public class Manager : Employee
{
// Managers need to know their number of stock options.
private ulong numberOfOptions;
public ulong NumbOpts
{
get { return numberOfOptions;}
set { numberOfOptions = value; }
}
}

public class SalesPerson : Employee
{
// Salespeople need to know their number of sales.
private int numberOfSales;
public int NumbSales
{
get { return numberOfSales;}
set { numberOfSales = value; }
}
}
}
```

Now that you have established an "is-a" relationship, `SalesPerson` and `Manager` have automatically inherited all public (and protected) members of the `Employee` base class. To illustrate:

```
// Create a subclass and access base class functionality.
static void Main(string[] args)
{
// Make a salesperson.
SalesPerson stan = new SalesPerson();
```

```
// These members are inherited from the Employee base class.
stan.ID = 100;
stan.Name = "Stan";

// This is defined by the SalesPerson class.
stan.NumbSales = 42;
Console.ReadLine();
}
```

Do be aware that inheritance preserves encapsulation. Therefore, a derived class cannot
directly access the private members defined by its base class.

Controlling Base Class Creation with base

Currently, `SalesPerson` and `Manager` can only be created using a default constructor. With this in mind, assume you have added a new six-argument constructor to the `Manager` type, which is invoked as follows:

```
static void Main(string[] args)
{
// Assume we now have the following constructor.
// (name, age, ID, pay, SSN, number of stock options).
Manager chucky = new Manager("Chucky", 35, 92, 100000, "333-23-2322", 9000);
}
```

If you look at the argument list, you can clearly see that most of these parameters should be
stored in the member variables defined by the `Employee` base class. To do so, you could = implement this new constructor as follows:

```
// If you do not say otherwise, a subclass constructor automatically calls the
// default constructor of its base class.
public Manager(string fullName, int age, int empID,
float currPay, string ssn, ulong numbOfOpts)
{
// This point of data belongs with us!
numberOfOptions = numbOfOpts;

// Leverage the various members inherited from Employee
// to assign the state data.
ID = empID;
Age = age;
Name = fullName;
SocialSecurityNumber = ssn;
Pay = currPay;
}
```

4.6 Keeping Family Secrets: The protected Keyword

As you already know, public items are directly accessible from anywhere, while private items cannot be accessed from any object beyond the class that has defined it. C#

takes the lead of many other modern object languages and provides an additional level of accessibility: protected. When a base class defines protected data or protected members, it is able to create a set of items that can be accessed directly by any descendent. If you wish to allow the `SalesPerson` and `Manager` child classes to directly access the data sector defined by `Employee`, you can update the original `Employee` class definition as follows:

```
// Protected state data.
public class Employee
{
// Child classes can directly access this information. Object users cannot.
protected string fullName;
protected int empID;
protected float currPay;
protected string empSSN;
protected int empAge;
...
}
```

The benefit of defining protected members in a base class is that derived types no longer have to access the data using public methods or properties. The possible downfall, of course, is that when a derived type has direct access to its parent's internal data, it is very possible to accidentally bypass existing business rules found within public properties (such as the mini-novel that exceeds the page count). When you define protected members, you are creating a level of trust between the parent and child class, as the compiler will not catch any violation of your type's business rules.Finally, understand that as far as the object user is concerned, protected data is regarded as private (as the user is "outside" of the family). Therefore, the following is illegal:

```
static void Main(string[] args)
{
// Error! Can't access protected data from object instance.
Employee emp = new Employee();
emp.empSSN = "111-11-1111";
}
```

4.7 The Third Pillar: C#'s Polymorphic Support

Let's now examine the final pillar of OOP: polymorphism. Recall that the `Employee` base class
defined a method named `GiveBonus()`, which was implemented as follows:

```
// Give bonus to employees.
public class Employee
{
...
public void GiveBonus(float amount)
{ currPay += amount; }
}
```

Because this method has been defined as public, you can now give bonuses to salespeople and managers (as well as part-time salespeople):

```
static void Main(string[] args)
{
// Give each employee a bonus.
Manager chucky = new Manager("Chucky", 50, 92, 100000, "333-23-2322", 9000);
chucky.GiveBonus(300);
chucky.DisplayStats();

SalesPerson fran = new SalesPerson("Fran", 43, 93, 3000, "932-32-3232", 31);
fran.GiveBonus(200);
fran.DisplayStats();
Console.ReadLine();
}
```

The problem with the current design is that the inherited `GiveBonus()` method operates identically for all subclasses. Ideally, the bonus of a salesperson or part-time salesperson should take into account the number of sales. Perhaps managers should gain additional stock options in conjunction with a monetary bump in salary. Given this, you are suddenly faced with an interesting question: "How can related objects respond differently to the same request?"

4.8 Casting Between.

Next up, you need to learn the laws of C# *casting operations*. Recall the `Employees` hierarchy and the fact that the topmost class in the system is `System.Object`. Therefore, everything "is-a" object and can be treated as such. Given this fact, it is legal to store an instance of any type within a object variable:

```
// A Manager "is-a" System.Object.
object frank = new Manager("Frank Zappa", 9, 40000, "111-11-1111", 5);
```

In the `Employees` system, `Managers`, `SalesPerson`, and `PTSalesPerson` types all extend `Employee`, so we can store any of these objects in a valid base class reference. Therefore, the following statements are also legal:

```
// A Manager "is-a" Employee too.
Employee moonUnit = new Manager("MoonUnit Zappa", 2, 20000, "101-11-1321", 1);

// A PTSalesPerson "is-a" SalesPerson.
SalesPerson jill = new PTSalesPerson("Jill", 834, 100000, "111-12-1119", 90);
```

The first law of casting between class types is that when two classes are related by an "is-a"
relationship, it is always safe to store a derived type within a base class reference. Formally, this is called an *implicit cast*, as "it just works" given the laws of inheritance. This leads to some powerful programming constructs. For example, if you have a class named `TheMachine` that supports the following static method:

```
public class TheMachine
{
public static void FireThisPerson(Employee e)
{
// Remove from database...
```

```
// Get key and pencil sharpener from fired employee...
}
}
```

you can effectively pass any descendent from the `Employee` class into this method directly, given the "is-a" relationship:

```
// Streamline the staff.
TheMachine.FireThisPerson(moonUnit);  // "moonUnit" was declared as an
Employee.
TheMachine.FireThisPerson(jill);      // "jill" was declared as a SalesPerson.
```

The following code compiles given the implicit cast from the base class type (`Employee`) to the derived type. However, what if you also wanted to fire Frank Zappa (currently stored in a generic `System.Object` reference)? If you pass the `frank` object directly into `TheMachine.FireThisPerson()` as follows:

```
// A Manager "is-a" object, but...
object frank = new Manager("Frank Zappa", 9, 40000, "111-11-1111", 5);
...
TheMachine.FireThisPerson(frank);     // Error!
```

you are issued a compiler error. The reason is you cannot automatically treat a `System.Object` as a derived `Employee` directly, given that `Object` "is-not-a" `Employee`. As you can see, however, the object reference is pointing to an `Employee`-compatible object. You can satisfy the compiler by performing an *explicit cast*.

In C#, explicit casts are denoted by placing parentheses around the type you wish to cast to, followed by the object you are attempting to cast from. For example:

```
// Cast from the generic System.Object into a strongly
// typed Manager.
Manager mgr = (Manager)frank;
Console.WriteLine("Frank's options: {0}", mgr.NumbOpts);
```

If you would rather not declare a specific variable of "type to cast to," you are able to condense the previous code as follows:

```
// An "inline" explicit cast.
Console.WriteLine("Frank's options: {0}",  ((Manager)frank).NumbOpts);
```

As far as passing the `System.Object` reference into the `FireThisPerson()` method, the problem can be rectified as follows:

```
// Explicitly cast System.Object into an Employee.
TheMachine.FireThisPerson((Employee)frank);
```

Recommended Questions:

1. What are Three Pillars of Object Oriented Programming?
2. Explain Encapsulation in c#?
3. List and Explain the various types of Inheritence.

UNIT - 5
Exceptions and Object Lifetime

5.1Ode to Errors, Bugs, and Exceptions,
5.2 The Role of .NET Exception Handing,
5.3 The System. Exception Base Class,
5.4 Throwing a Generic Exception,
5.5 Catching Exception,
5.6 CLR System – Level Exception (System. System Exception),
5.7 Custom Application-Level Exception (System. System Exception),
5.8 Handling Multiple Exception
5.9 The Fanily Block, the Last Chance Exception Dynamically
Identifying Application – and System Level Exception Debugging System
5.10 Exception Using VS. NET,
5.11 Understanding Object Lifetime,
5.12 The CIT of "new',
5.13 The Basics of Garbage Collection,
5.14 Finalization a Type, The Finalization Process,
5.15 Building an Ad Hoc Destruction Method,
5.16 Garbage Collection Optimizations,
5.17 The System. GC Type.

5.1 Ode to Errors, Bugs, and Exceptions

Three commonly used anomaly-centric terms:

• *Bugs*: This is, simply put, an error on the part of the programmer. For example, assume you are programming with unmanaged C++. If you make calls on a `NULL` pointer or fail to delete allocated memory (resulting in a memory leak), you have a bug.

• *User errors*: Unlike bugs, user errors are typically caused by the individual running your application, rather than by those who created it. For example, an end user who enters a malformed string into a text box could very well generate an error *if* you fail to handle this faulty input in your code base.

• *Exceptions*: Exceptions are typically regarded as runtime anomalies that are difficult, if not impossible, to account for while programming your application. Possible exceptions include attempting to connect to a database that no longer exists, opening a corrupted file, or contacting a machine that is currently offline. In each of these cases, the programmer (and end user) has little control over these "exceptional" circumstances. Given the previous definitions, it should be clear that .NET structured *exception* handling is a technique well suited to deal with runtime *exceptions*. However, as for the bugs and user errors that have escaped your view, the CLR will often generate a corresponding exception that identi- fies the problem at hand. The .NET base class libraries define numerous exceptions such as `FormatException`, `IndexOutOfRangeException`, `FileNotFoundException`, `ArgumentOutOfRangeException`, and so forth.

5.2 The Role of .NET Exception Handling

Prior to .NET, error handling under the Windows operating system was a confused mishmash of techniques. Many programmers rolled their own error handling logic within the context of a given application. For example, a development team may define a set of numerical constants that represent known error conditions, and make use of them as method return values. By way of an example, ponder the following partial C code:

```
/* A very C-style error trapping mechanism. */
#define E_FILENOTFOUND 1000

int SomeFunction()
{
// Assume something happens in this f(x)
// that causes the following return value.
return E_FILENOTFOUND;
}

void main()
{
int retVal = SomeFunction();
if(retVal == E_FILENOTFOUND)
printf("Cannot find file...");
```

```
}
```

This approach is less than ideal, given the fact that the constant `E_FILENOTFOUND` is little more than a numerical value, and is far from being a helpful agent regarding how to deal with the problem. Ideally, you would like to wrap the name, message, and other helpful information regarding this error condition into a single, well-defined package (which is exactly what happens under structured exception handling). In addition to a developer's ad hoc techniques, the Windows API defines hundreds of error codes that come by way of `#defines`, `HRESULTs`, and far too many variations on the simple Boolean (`bool`, `BOOL`, `VARIANT_BOOL`, and so on). Also, many C++ COM developers (and indirectly, many VB6 COM developers) have made use of a small set of standard COM interfaces (e.g., `ISupportErrorInfo`, `IErrorInfo`, `ICreateErrorInfo`) to return meaningful error information to a COM client. The obvious problem with these previous techniques is the tremendous lack of symmetry. Each approach is more or less tailored to a given technology, a given language, and perhaps even a given project. In order to put an end to this madness, the .NET platform provides a standard technique to send and trap runtime errors: structured exception handling (SEH).

The Atoms of .NET Exception Handling

Programming with structured exception handling involves the use of four interrelated entities:

- A class type that represents the details of the exception that occurred
- A member that *throws* an instance of the exception class to the caller
- A block of code on the caller's side that invokes the exception-prone member
- A block of code on the caller's side that will process (or *catch*) the exception should it occur

The C# programming language offers four keywords (`try`, `catch`, `throw`, and `finally`) that allow you to throw and handle exceptions. The type that represents the problem at hand is a class derived from `System.Exception` (or a descendent thereof). Given this fact, let's check out the role of this exception-centric base class.

5.3 The System.Exception Base Class

All user- and system-defined exceptions ultimately derive from the `System.Exception` base class (which in turn derives from `System.Object`). Note that some of these members are virtual and may thus be overridden by derived types:

```
public class Exception : ISerializable, _Exception
{
public virtual IDictionary Data { get; }
protected Exception(SerializationInfo info, StreamingContext context);
public Exception(string message, Exception innerException);
public Exception(string message);
public Exception();
public virtual Exception GetBaseException();
public virtual void GetObjectData(SerializationInfo info,
StreamingContext context);
public System.Type GetType();
protected int HResult { get; set; }
```

```
public virtual string HelpLink { get; set; }
public System.Exception InnerException { get; }
public virtual string Message { get; }
public virtual string Source { get; set; }
public virtual string StackTrace { get; }
public MethodBase TargetSite { get; }
public override string ToString();
}
```

The Simplest Possible Example

To illustrate the usefulness of structured exception handling, we need to create a type that may throw an exception under the correct circumstances. Assume we have created a new console application project (named SimpleException) that defines two class types (`Car` and `Radio`) associated using the "has-a" relationship. The `Radio` type defines a single method that turns the radio's power on or off:

```
public class Radio
{
public void TurnOn(bool on)
{
if(on)
Console.WriteLine("Jamming...");
else
Console.WriteLine("Quiet time...");
}
}
```

5.4 Throwing a Generic Exception

Now that we have a functional `Car` type, I'll illustrate the simplest way to throw an exception. The current implementation of `Accelerate()` displays an error message if the caller attempts to speed up the `Car` beyond its upper limit. To retrofit this method to throw an exception if the user attempts to speed up the automobile after it has met its maker, you want to create and configure a new instance of the `System.Exception` class, setting the value of the read-only `Message` property via the class constructor. When you wish to send the error object back to the caller, make use of the C# `throw` keyword. Here is the relevant code update to the `Accelerate()` method:

```
// This time, throw an exception if the user speeds up beyond maxSpeed.
public void Accelerate(int delta)
{
if (carIsDead)
Console.WriteLine("{0} is out of order...", petName);
else
{
currSpeed += delta;
if (currSpeed >= maxSpeed)
{
carIsDead = true;
currSpeed = 0;
// Use "throw" keyword to raise an exception.
```

```
throw new Exception(string.Format("{0} has overheated!", petName));
}
else
Console.WriteLine("=> CurrSpeed = {0}", currSpeed);
}
}
```

Before examining how a caller would catch this exception, a few points of interest. First of all, when you are throwing an exception, it is always up to you to decide exactly what constitutes the error in question, and when it should be thrown. Here, you are making the assumption that if the program attempts to increase the speed of a car that has expired, a `System.Exception` type should be thrown to indicate the `Accelerate()` method cannot continue (which may or may not be a valid assumption). Alternatively, you could implement `Accelerate()` to recover automatically without needing to throw an exception in the first place. By and large, exceptions should be thrown only when a more terminal condition has been met (for example, not finding a necessary file, failing to connect to
a database, and whatnot). Deciding exactly what constitutes throwing an exception is a design issue you must always contend with. For our current purposes, assume that asking a doomed automobile to increase its speed justifies a cause to throw an exception.

5.5 Catching Exceptions

Because the `Accelerate()` method now throws an exception, the caller needs to be ready to handle the exception should it occur. When you are invoking a method that may throw an exception, you make use of a `try/catch` block. Once you have caught the exception type, you are able to invoke the members of the `System.Exception` type to extract the details of the problem. What you do with this data is largely up to you. You may wish to log this information to a report file, write the data to the Windows event log, e-mail a system administrator, or display the problem to the end user. Here, you will simply dump the contents to the console window:

```
// Handle the thrown exception.
static void Main(string[] args)
{
Console.WriteLine("***** Creating a car and stepping on it *****");
Car myCar = new Car("Zippy", 20);
myCar.CrankTunes(true);

// Speed up past the car's max speed to
// trigger the exception.
try
{
for(int i = 0; i < 10; i++)
myCar. Accelerate(10);
}
catch(Exception e)
{
```

```
Console.WriteLine("\n*** Error! ***");
Console.WriteLine("Method: {0}", e.TargetSite);
Console.WriteLine("Message: {0}", e.Message);
Console.WriteLine("Source: {0}", e.Source);
}

// The error has been handled, processing continues with the next statement.
Console.WriteLine("\n***** Out of exception logic *****");
Console.ReadLine();
}
```

5.6 CLR System-Level Exceptions (System.SystemException)

The .NET base class libraries define many classes derived from `System.Exception`. For example, the `System` namespace defines core error objects such as `ArgumentOutOfRangeException`, `IndexOutOfRangeException`, `StackOverflowException`, and so forth. Other namespaces define exceptions that reflect the behavior of that namespace (e.g., `System.Drawing.Printing` defines printing exceptions, `System.IO` defines IO-based exceptions, `System.Data` defines database-centric exceptions, and so forth). Exceptions that are thrown by the CLR are (appropriately) called *system exceptions*. These exceptions are regarded as nonrecoverable, fatal errors. System exceptions derive directly from a base class named `System.SystemException`, which in turn derives from `System.Exception` (which derives from `System.Object`):

```
public class SystemException : Exception
{
// Various constructors.
}
```

Given that the `System.SystemException` type does not add any additional functionality beyond a set of constructors, you might wonder why `SystemException` exists in the first place. Simply put, when an exception type derives from `System.SystemException`, you are able to determine that the .NET runtime is the entity that has thrown the exception, rather than the code base of the executing application.

5.7 Custom Application-Level Exceptions (System.ApplicationException)

Given that all .NET exceptions are class types, you are free to create your own application-specific exceptions. However, due to the fact that the `System.SystemException` base class represents exceptions thrown from the CLR, you may naturally assume that you should derive your custom exceptions from the `System.Exception` type. While you could do so, best practice dictates that you instead derive from the `System.ApplicationException` type:

```
public class ApplicationException : Exception
{
// Various constructors.
}
```

Building Custom Exceptions, Take One

While you can always throw instances of `System.Exception` to signal a runtime error (as shown in our first example), it is sometimes advantageous to build a *strongly typed*

exception that represents the unique details of your current problem. For example, assume you wish to build a custom exception (named `CarIsDeadException`) to represent the error of speeding up a doomed automobile. The first step is to derive a new class from `System.ApplicationException` (by convention, all exception classes end with the "Exception" suffix).

```
// This custom exception describes the details of the car-is-dead condition.
public class CarIsDeadException : ApplicationException
{}
```

Like any class, you are free to include any number of custom members that can be called within the `catch` block of the calling logic. You are also free to override any virtual members defined by your parent classes. For example, we could implement `CarIsDeadException` by overriding the virtual `Message` property:

```
public class CarIsDeadException : ApplicationException
{
private string messageDetails;

public CarIsDeadException(){ }
public CarIsDeadException(string message)
{
messageDetails = message;
}

// Override the Exception.Message property.
public override string Message
{
get
{
return string.Format("Car Error Message: {0}", messageDetails);
}
}
}
```

Here, the `CarIsDeadException` type maintains a private data member (`messageDetails`) that represents data regarding the current exception, which can be set using a custom constructor. Throwing this error from the `Accelerate()` is straightforward. Simply allocate, configure, and throw a `CarIsDeadException` type rather than a generic `System.Exception`:

```
// Throw the custom CarIsDeadException.
public void Accelerate(int delta)
{
...
CarIsDeadException ex = new CarIsDeadException(string.Format("{0} has
overheated!", petName));
ex.HelpLink = "http://www.CarsRUs.com";
ex.Data.Add("TimeStamp", string.Format("The car exploded at {0}",
DateTime.Now));
ex.Data.Add("Cause", "You have a lead foot.");
```

```
throw ex;
...
}
```

To catch this incoming exception explicitly, your `catch` scope can now be updated to catch a specific `CarIsDeadException` type (however, given that `CarIsDeadException` "is-a" `System.Exception`, it is still permissible to catch a generic `System.Exception` as well):

```
static void Main(string[] args)
{
...
catch(CarIsDeadException e)
{
// Process incoming exception.
}
...
}
```

So, now that you understand the basic process of building a custom exception, you may wonder when you are required to do so. Typically, you only need to create custom exceptions when the error is tightly bound to the class issuing the error (for example, a custom `File` class that throws a number of file-related errors, a `Car` class that throws a number of car-related errors, and so forth). In doing so, you provide the caller with the ability to handle numerous exceptions on an error-by-error basis.

5.8 Handling Multiple Exceptions

In its simplest form, a `try` block has a single `catch` block. In reality, you often run into a situation where the statements within a `try` block could trigger *numerous* possible exceptions. For example, assume the car's `Accelerate()` method also throws a base-class-library predefined `ArgumentOutOfRangeException` if you pass an invalid parameter (which we will assume is any value less than zero):

```
// Test for invalid argument before proceeding.
public void Accelerate(int delta)
{
if(delta < 0)
throw new ArgumentOutOfRangeException("Speed must be greater than zero!");
...
}
```

The `catch` logic could now specifically respond to each type of exception:

```
static void Main(string[] args)
{
...
// Here, we are on the lookout for multiple exceptions.
try
{
for(int i = 0; i < 10; i++)
myCar.Accelerate(10);
}
```

```
catch(CarIsDeadException e)
{
// Process CarIsDeadException.
}
catch(ArgumentOutOfRangeException e)
{
// Process ArgumentOutOfRangeException.
}
...
}
```

When you are authoring multiple `catch` blocks, you must be aware that when an exception is thrown, it will be processed by the "first available" catch. To illustrate exactly what the "first available" catch means, assume you retrofitted the previous logic with an addition `catch` scope that attempts to handle all exceptions beyond `CarIsDeadException` and `ArgumentOutOfRangeException` by catching a generic `System.Exception` as follows:

```
// This code will not compile!
static void Main(string[] args)
{
...
try
{
for(int i = 0; i < 10; i++)
myCar.Accelerate(10);
}
catch(Exception e)
{
// Process all other exceptions?
}
catch(CarIsDeadException e)
{
// Process CarIsDeadException.
}
catch(ArgumentOutOfRangeException e)
{
// Process ArgumentOutOfRangeException.
}
...
}
```

Thus, if you wish to define a `catch` statement that will handle any errors beyond `CarIsDeadException` and `ArgumentOutOfRangeException`, you would write the following:

```
// This code compiles just fine.
static void Main(string[] args)
{
...
try
```

```
{
for(int i = 0; i < 10; i++)
myCar.Accelerate(10);
}
catch(CarIsDeadException e)
{
// Process CarIsDeadException.
}
catch(ArgumentOutOfRangeException e)
{
// Process ArgumentOutOfRangeException.
}
catch(Exception e)
{
// This will now handle all other possible exceptions
// thrown from statements within the try scope.
}
...
}
```

Generic catch Statements

C# also supports a "generic" `catch` scope that does not explicitly receive the exception object thrown by a given member:

```
// A generic catch.
static void Main(string[] args)
{
...
try
{
for(int i = 0; i < 10; i++)
myCar.Accelerate(10);
}
catch
{
Console.WriteLine("Something bad happened...");
}
...
}
```

Obviously, this is not the most informative way to handle exceptions, given that you have no way to obtain meaningful data about the error that occurred (such as the method name, call stack, or custom message). Nevertheless, C# does allow for such a construct.

Rethrowing Exceptions

Be aware that it is permissible for logic in a `try` block to *rethrow* an exception up the call stack to the previous caller. To do so, simply make use of the `throw` keyword within

a `catch` block. This passes the exception up the chain of calling logic, which can be helpful if your `catch` block is only able to partially handle the error at hand:

```
// Passing the buck.
static void Main(string[] args)
{
...
try
{
// Speed up car logic...
}
catch(CarIsDeadException e)
{
// Do any partial processing of this error and pass the buck.
// Here, we are rethrowing the incoming CarIsDeadException object.
// However, you are also free to throw a different exception if need be.
throw e;
}
...
}
```

5.9 The Finally Block

A `try/catch` scope may also define an optional `finally` block. The motivation behind a `finally` block is to ensure that a set of code statements will *always* execute, exception (of any type) or not. To illustrate, assume you wish to always power down the car's radio before exiting `Main()`, regardless of any handled exception:

```
static void Main(string[] args)
{
...
Car myCar = new Car("Zippy", 20);
myCar.CrankTunes(true);

try
{
// Speed up car logic.
}
catch(CarIsDeadException e)
{
// Process CarIsDeadException.
}
catch(ArgumentOutOfRangeException e)
{
// Process ArgumentOutOfRangeException.
}
catch(Exception e)
{
// Process any other Exception.
```

```
}
finally
{
// This will always occur. Exception or not.
myCar.CrankTunes(false);
}
...
}
```

5.10 Exceptions Using Visual Studio 2005

To wrap things up, do be aware that Visual Studio 2005 provides a number of tools that help you debug unhandled custom exceptions. Again, assume you have increased the speed of a `Car` object beyond the maximum. If you were to start a debugging session (using the Debug Start menu selection), Visual Studio automatically breaks at the time the uncaught exception is thrown. Better yet, you are presented with a window.

5.11 Understanding Object Lifetime

When you are building your C# applications, you are correct to assume that the managed heap will take care of itself without your direct intervention. In fact, the golden rule of .NET memory management is simple:

• **Rule:** Allocate an object onto the managed heap using the `new` keyword and forget about it. Once "`new`-ed," the garbage collector will destroy the object when it is no longer needed. The next obvious question, of course, is, "How does the garbage collector determine when an object is no longer needed"? The short (i.e., incomplete) answer is that the garbage collector removes an object from the heap when it is *unreachable* by any part of your code base. Assume you have a method that allocates a local `Car` object:

```
public static void MakeACar()
{
// If myCar is the only reference to the Car object,
// it may be destroyed when the method returns.
Car myCar = new Car();
...
}
```

5.12The CIL of new

When the C# compiler encounters the `new` keyword, it will emit a CIL `newobj` instruction into the method implementation. If you were to compile the current example code and investigate the resulting assembly using `ildasm.exe`, you would find the following CIL statements within the `MakeACar()` method:

```
.method public hidebysig static void MakeACar() cil managed
{
// Code size 7 (0x7)
.maxstack 1
.locals init ([0] class SimpleFinalize.Car c)
IL_0000: newobj instance void SimpleFinalize.Car::.ctor()
IL_0005: stloc.0
IL_0006: ret
} // end of method Program::MakeACar
```

Before we examine the exact rules that determine when an object is removed from the managed heap, let's check out the role of the CIL `newobj` instruction in a bit more detail. First, understand that the managed heap is more than just a random chunk of memory accessed by the CLR. The .NET garbage collector is quite a tidy housekeeper of the heap, given that it will compact empty blocks of memory (when necessary) for purposes of optimization. To aid in this endeavor, the managed heap maintains a pointer (commonly referred to as the *next object pointer* or *new object pointer*) that identifies exactly where the next object will be located. These things being said, the `newobj` instruction informs the CLR to perform the following core tasks:

- Calculate the total amount of memory required for the object to be allocated (including the necessary memory required by the type's member variables and the type's base classes).
- Examine the managed heap to ensure that there is indeed enough room to host the object to be allocated. If this is the case, the type's constructor is called, and the caller is ultimately returned a reference to the new object in memory, whose address just happens to be identical to the last position of the next object pointer.
- Finally, before returning the reference to the caller, advance the next object pointer to point to the next available slot on the managed heap.

As you are busy allocating objects in your application, the space on the managed heap may eventually become full. When processing the `newobj` instruction, if the CLR determines that the managed heap does not have sufficient memory to allocate the requested type, it will perform a garbage collection in an attempt to free up memory. Thus, the next rule of garbage collection is also quite simple.

- **Rule:** If the managed heap does not have sufficient memory to allocate a requested object, a garbage collection will occur. When a collection does take place, the garbage collector temporarily suspends all active *threads* within the current process to ensure that the application does not access the heap during the collection process. However, for the time being, simply regard a thread as a path of execution within a running executable. Once the garbage collection cycle has completed, the suspended threads are permitted to carry on their work. Thankfully, the .NET garbage collector is highly optimized; you will seldom (if ever) notice this brief interruption in your application.

5.13 The basics of Garbage collection

When the CLR is attempting to locate unreachable objects, is does *not* literally examine each and every object placed on the managed heap. Obviously, doing so would involve considerable time, especially in larger (i.e., real-world) applications. To help optimize the process, each object on the heap is assigned to a specific "generation."The idea behind generations is simple: The longer an object has existed on the heap, the more likely it is to stay there. For example, the object implementing `Main()` will be in memory until the program terminates. Conversely, objects that have been recently placed on the heap are likely to be unreachable rather quickly (such as an object created within a method scope). Given these assumptions, each object on the heap belongs to one of the following generations:

- *Generation 0*: Identifies a newly allocated object that has never been marked for collection

• *Generation 1*: Identifies an object that has survived a garbage collection (i.e., it was marked for collection, but was not removed due to the fact that the sufficient heap space was acquired)
• *Generation 2*: Identifies an object that has survived more than one sweep of the garbage collector The garbage collector will investigate all generation 0 objects first. If marking and sweeping these objects results in the required amount of free memory, any surviving objects are promoted to generation 1. To illustrate how an object's generation affects the collection process.

5.14 The Finalization a Type, The Finalization Process

Not to beat a dead horse, but always remember that the role of the `Finalize()` method is to ensure that a .NET object can clean up unmanaged resources when garbage collected. Thus, if you are building a type that does not make use of unmanaged entities (by far the most common case), finalization is of little use. In fact, if at all possible, you should design your types to avoid supporting a`Finalize()` method for the very simple reason that finalization takes time. When you allocate an object onto the managed heap, the runtime automatically determines whether your object supports a custom `Finalize()` method. If so, the object is marked as *finalizable*, and a pointer to this object is stored on an internal queue named the *finalization queue*. The finalization queue is a table maintained by the garbage collector that points to each and every object that must be finalized before it is removed from the heap. When the garbage collector determines it is time to free an object from memory, it examines each entry on the finalization queue, and copies the object off the heap to yet another managed structure termed the *finalization reachable* table (often abbreviated as freachable, and pronounced "eff-reachable"). At this point, a separate thread is spawned to invoke the `Finalize()` method for each object on the freachable table *at the next garbage collection*. Given this, it will take at very least *two* garbage collections to truly finalize an object. The bottom line is that while finalization of an object does ensure an object can clean up unmanaged resources, it is still nondeterministic in nature, and due to the extra behind-the- curtains processing, considerably slower.

5.15 Building Ad Hoc DestructionMethod

The supreme base class of .NET, `System.Object`, defines a virtual method named `Finalize()`. The default implementation of this method does nothing whatsoever:

```
// System.Object
public class Object
{
...
protected virtual void Finalize() {}
}
```

When you override `Finalize()` for your custom classes, you establish a specific location to perform any necessary cleanup logic for your type. Given that this member is defined as protected, it is not possible to directly call an object's `Finalize()` method. Rather, the *garbage collector* will call an object's `Finalize()` method (if supported) before removing the object from memory. Of course, a call to `Finalize()` will

(eventually) occur during a "natural" garbage collection or when you programmatically force a collection via `GC.Collect()`. In addition, a type's finalizer method will automatically be called when the *application domain* hosting your application is unloaded from memory. Based on your current background in .NET, you may know that application domains (or simply AppDomains) are used to host an executable assembly and any necessary external code libraries.

5.16 Detailing the Finalization Process

Not to beat a dead horse, but always remember that the role of the `Finalize()` method is to ensure that a .NET object can clean up unmanaged resources when garbage collected. Thus, if you are building a type that does not make use of unmanaged entities (by far the most common case), finalization is of little use. In fact, if at all possible, you should design your types to avoid supporting a`Finalize()` method for the very simple reason that finalization takes time. When you allocate an object onto the managed heap, the runtime automatically determines whether your object supports a custom `Finalize()` method. If so, the object is marked as *finalizable*, and a pointer to this object is stored on an internal queue named the *finalization queue*. The finalizetion queue is a table maintained by the garbage collector that points to each and every object that must be finalized before it is removed from the heap. When the garbage collector determines it is time to free an object from memory, it examines each entry on the finalization queue, and copies the object off the heap to yet another managed structure termed the *finalization reachable* table (often abbreviated as freachable, and pronounced "eff-reachable"). At this point, a separate thread is spawned to invoke the `Finalize()` method for each object on the freachable table *at the next garbage collection*. Given this, it will take at very least *two* garbage collections to truly finalize an object. The bottom line is that while finalization of an object does ensure an object can clean up unmanaged resources, it is still nondeterministic in nature, and due to the extra behind-the- curtains processing, considerably slower.

5.17 The System.GC Type

The base class libraries provide a class type named `System.GC` that allows you to programmatically interact with the garbage collector using a set of static members. Now, do be very aware that you will seldom (if ever) need to make use of this type directly in your code. Typically speaking, the only time you will make use of the members of `System.GC` is when you are creating types that make use of *unmanaged resources*. Table 5-1 provides a rundown of some of the more interesting members (consult the .NET Framework 2.0 SDK Documentation for complete details).

Table 5-1. *Select Members of the* `System.GC` *Type*

`System.GC` **Member**	**Meaning in Life**
`AddMemoryPressure()`	Allow you to specify a numerical value that represents the calling object's "urgency level" regarding the garbage collection process. Be aware that these methods should alter pressure *in tandem*

	and thus added.
`Collect()`	Forces the GC to perform a garbage collection.
`CollectionCount()`	Returns a numerical value representing how many times a given generation has been swept.
`GetGeneration()`	Returns the generation to which an object currently belongs.
`GetTotalMemory()`	Returns the estimated amount of memory (in bytes) currently allocated on the managed heap. The Boolean parameter specifies whether the call should wait for garbage collection to occur before returning.
`MaxGeneration`	system. Under Microsoft's .NET 2.0, there are three possible generations (0, 1, and 2).
`SuppressFinalize()`	Sets a flag indicating that the specified object should not have its `Finalize()` method called.
`WaitForPendingFinalizers()`	Suspends the current thread until all finalizable objects have been finalized. This method is typically called directly after invoking

Ponder the following `Main()` method, which illustrates select members of `System.GC`:

```
static void Main(string[] args)
{
// Print out estimated number of bytes on heap.
Console.WriteLine("Estimated bytes on heap: {0}",
GC.GetTotalMemory(false));

// MaxGeneration is zero based, so add 1 for display purposes.
Console.WriteLine("This OS has {0} object generations.\n",
(GC.MaxGeneration + 1));

Car refToMyCar = new Car("Zippy", 100);
Console.WriteLine(refToMyCar.ToString());

// Print out generation of refToMyCar object.
Console.WriteLine("Generation of refToMyCar is: {0}",
GC.GetGeneration(refToMyCar));

Console.ReadLine();
}
```

Recommended Questions:

1. Explain the process of finalize object in .net environment
2. Write a program in c# to throw and handle following exceptions in banking application minimum balance exception argument out of range exception
3. List and explain with code core members of system. Exception type
4. Define a method that would and sort an array of integer
5. List and explain core members of the system exception type.ow would you build custom exception?
6. Write c# application to illustrate handling multiple exceptions.
7. What is meant by object life time? Describe the role of .Net garbase collection, finalization process and Ad Hoc destruction method, with examples.

UNIT - 6
Interfaces and Collections

6.1 Defining Interfaces Using C#
6.2 Invoking Interface Members at the object Level,
6.3 Exercising the Shapes Hierarchy
6.4 Understanding Explicit Interface Implementation,
6.5 Interfaces As Polymorphic Agents,
6.6 Building Interface Hierarchies Implementation,
6.7 Interfaces Using VS .NET,understanding the IConvertible Interface,
6.8 Building a Custom Enumerator (IEnumerable and Enumerator),
6.9 Building Clone able objects (ICloneable),
6.10 Building Comparable Objects I Comparable
6.11 Exploring the system. Collections Namespace
6.12 Building a Custom Container (Retrofitting the Cars Type).

6.1 Defining Interfaces in C#

To begin this chapter, allow me to provide a formal definition of the "interface" type. An interface is nothing more than a named collection of semantically related *abstract members*. The specific members defined by an interface depend on the exact *behavior* it is modeling. Yes, it's true. An interface expresses a behavior that a given class or structure may choose to support. At a syntactic level, an interface is defined using the C# `interface` keyword. Unlike other .NET types, interfaces never specify a base class (not even `System.Object`) and contain members that do *not* take an access modifier (as all interface members are implicitly public). To get the ball rolling, here is a custom interface defined in C#:

```
// This interface defines the behavior of "having points."
public interface IPointy
{
// Implicitly public and abstract.
byte GetNumberOfPoints();
}
```

As you can see, the `IPointy` interface defines a single method. However, .NET interface types are also able to define any number of properties. For example, you could create the `IPointy` interface to use a read-only property rather than a traditional accessor method:

```
// The pointy behavior as a read-only property.
public interface IPointy
{
byte Points{get;}
}
```

Do understand that interface types are quite useless on their own, as they are nothing more than a named collection of abstract members. Given this, you cannot allocate interface types as you would a class or structure:

```
// Ack! Illegal to "new" interface types.
static void Main(string[] args)
{
IPointy p = new IPointy(); // Compiler error!
}
```

Interfaces do not bring much to the table until they are implemented by a class or structure. Here, `IPointy` is an interface that expresses the behavior of "having points." As you can tell, this behavior might be useful in the shapes hierarchy developed in Chapter 4. The idea is simple: Some classes in the Shapes hierarchy have points (such as the `Hexagon`), while others (such as the `Circle`) do not. If you configure `Hexagon` and `Triangle` to implement the `IPointy` interface, you can safely assume that each class now supports a common behavior, and therefore a common set of members.

6.2 Invoking Interface Members at object level in C#

When a class (or structure) chooses to extend its functionality by supporting interface types, it does so using a comma-delimited list in the type definition. Be aware that the direct base class must be the *first item* listed after the colon operator. When your class

type derives directly from System.Object, you are free to simply list the interface(s) supported by the class, as the C# compiler will extend your types from System.Object if you do not say otherwise. On a related note, given that structures always derive from System.ValueType (see Chapter 3), simply list each interface directly after the structure definition. Ponder the following examples:

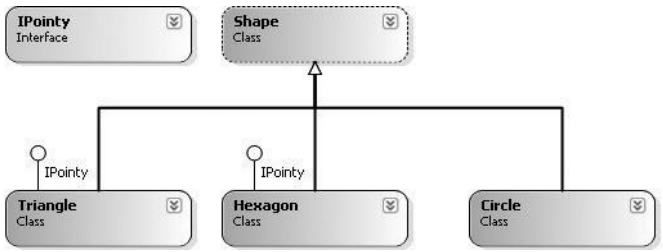
```
// This class derives from System.Object and
// implements a single interface.
public class SomeClass : ISomeInterface
{...}

// This class also derives from System.Object
// and implements a single interface.
public class MyClass : object, ISomeInterface
{...}

// This class derives from a custom base class
// and implements a single interface.
public class AnotherClass : MyBaseClass, ISomeInterface
{...}

// This struct derives from System.ValueType and
// implements two interfaces.
public struct SomeStruct : ISomeInterface, IPointy
{...}
```

Understand that implementing an interface is an all-or-nothing proposition. The supporting type is not able to selectively choose which members it will implement. Given that the IPointy interface defines a single property, this is not too much of a burden. However, if you are implementing an interface that defines ten members, the type is now responsible for fleshing out the details of the ten abstract entities. In any case, here is the implementation of the updated shapes hierarchy (note the new Triangle class type):

```
// Hexagon now implements IPointy.
public class Hexagon : Shape, IPointy
{
public Hexagon(){ }
public Hexagon(string name) : base(name){ }
public override void Draw()
{ Console.WriteLine("Drawing {0} the Hexagon", PetName); }

// IPointy Implementation.
public byte Points
{
get { return 6; }
}
}

// New Shape derived class named Triangle.
```

```
public class Triangle : Shape, IPointy
{
public Triangle() { }
public Triangle(string name) : base(name) { }
public override void Draw()
{ Console.WriteLine("Drawing {0} the Triangle", PetName); }

// IPointy Implementation.
public byte Points
{
get { return 3; }
}
}
```

6.3 Excercising Shape Hierarchy



6.4Understanding Explicit Interface Implementation

In our current definition of `IDraw3D`, we were forced to name its sole method `Draw3D()` in order to avoid clashing with the abstract `Draw()` method defined in the `Shape` base class. While there is nothing horribly wrong with this interface definition, a more natural method name would simply be `Draw()`:

```
// Refactor method name from "Draw3D" to "Draw".
public interface IDraw3D
{
void Draw();
}
```

If we were to make such a change, this would require us to also update our implementation of `DrawIn3D()`.

```
public static void DrawIn3D(IDraw3D itf3d)
{
Console.WriteLine("-> Drawing IDraw3D compatible type");
itf3d.Draw();
}
```

Now, assume you have defined a new class named `Line` that derives from the abstract `Shape`
class and implements `IDraw3D` (both of which now define an identically named abstract `Draw()` method):

```
// Problems? It depends...
public class Line : Shape, IDraw3D
{
public override void Draw()
{
Console.WriteLine("Drawing a line...");
}
}
```

6.5 Interface as Polymorphic Agent

Explicit interface implementation can also be very helpful whenever you are implementing a number of interfaces that happen to contain identical members. For example, assume you wish to create a class that implements all the following new interface types:

```
// Three interfaces each define identically named methods.
public interface IDraw
{
void Draw();
}

public interface IDrawToPrinter
{
void Draw();
}
```

If you wish to build a class named `SuperImage` that supports basic rendering (`IDraw`), 3D rendering (`IDraw3D`), as well as printing services (`IDrawToPrinter`), the only way to provide unique implementations for each method is to use explicit interface implementation:

```
// Not deriving from Shape, but still injecting a name clash.
public class SuperImage : IDraw, IDrawToPrinter, IDraw3D
{
void IDraw.Draw()
{ /* Basic drawing logic. */ }

void IDrawToPrinter.Draw()
{ /* Printer logic. */ }

void IDraw3D.Draw()
{ /* 3D rendering logic. */ }
}
```

6.6 Building Interface Hierarchies

To continue our investigation of creating custom interfaces, let's examine the topic of *interface hierarchies*. Just as a class can serve as a base class to other classes (which can in turn function as base classes to yet another class), it is possible to build inheritance relationships among interfaces. As you might expect, the topmost

interface defines a general behavior, while the most derived interface defines more specific behaviors. To illustrate, ponder the following interface hierarchy:

```
// The base interface.
public interface IDrawable
{  void Draw();}

public interface IPrintable : IDrawable
{  void Print(); }

public interface IMetaFileRender : IPrintable
{  void Render(); }
```

Now, if a class wished to support each behavior expressed in this interface hierarchy, it would derive from the *nth-most* interface (`IMetaFileRender` in this case). Any methods defined by the base interface(s) are automatically carried into the definition. For example:

```
// This class supports IDrawable, IPrintable, and IMetaFileRender.
public class SuperImage : IMetaFileRender
{
public void Draw()
{ Console.WriteLine("Basic drawing logic."); }

public void Print()
{ Console.WriteLine("Draw to printer."); }

public void Render()
{ Console.WriteLine("Render to metafile."); }
}
```

Here is some sample usage of exercising each interface from a `SuperImage` instance:

```
// Exercise the interfaces.
static void Main(string[] args)
{
SuperImage si = new SuperImage();

// Get IDrawable.
IDrawable itfDraw = (IDrawable)si;
itfDraw.Draw();

// Now get ImetaFileRender, which exposes all methods up
// the chain of inheritance.
if (itfDraw is IMetaFileRender)
{
IMetaFileRender itfMF = (IMetaFileRender)itfDraw;
itfMF.Render();
itfMF.Print();
}
Console.ReadLine();
}
```

6.7 Interfaces Using VS.NET,Understanding the IConvertable Interface

Although interface-based programming is a very powerful programming technique, implementing interfaces may entail a healthy amount of typing. Given that interfaces are a named set of abstract members, you will be required to type in the stub code (and implementation) for *each* interface method on *each* class that supports the behavior.

As you would hope, Visual Studio 2005 does support various tools that make the task of implementing interfaces less burdensome. Assume you wish to implement the `ICar` interface on a new class named `MiniVan`. You will notice when you complete typing the interface's name (or when you position the mouse cursor on the interface name in the code window), the first letter is underlined Once you select options, you will see that Visual Studio 2005 has built generated stub code (within a named code region) for you to update (note that the default implementation throws a `System.Exception`).

```
namespace IFaceHierarchy
{
public class MiniVan : ICar
{
public MiniVan()
{
}

#region ICar Members
public void Drive()
{
new Exception("The method or operation is not implemented.");
}
#endregion
}
}
```

Now that you have drilled into the specifics of building and implementing custom interfaces, the remainder of the chapter examines a number of predefined interfaces contained within the .NET base class libraries.

6.8Building Enumerable Types (IEnumerable and IEnumerator)

To illustrate the process of implementing existing .NET interfaces, let's first examine the role of `IEnumerable` and `IEnumerator`. Assume you have developed a class named `Garage` that contains a set of individual `Car` types (see Chapter 6) stored within a `System.Array`:

```
// Garage contains a set of Car objects.
public class Garage
{
private Car[] carArray;

// Fill with some Car objects upon startup.
```

```
public Garage()
{
carArray = new Car[4];
carArray[0] = new Car("Rusty", 30);
carArray[1] = new Car("Clunker", 55);
carArray[2] = new Car("Zippy", 30);
carArray[3] = new Car("Fred", 30);
}
}
```

Ideally, it would be convenient to iterate over the `Garage` object's subitems using the C# `foreach` construct:

```
// This seems reasonable...
public class Program
{
static void Main(string[] args)
{
Garage carLot = new Garage();

// Hand over each car in the collection?
foreach (Car c in carLot)
{
Console.WriteLine("{0} is going {1} MPH",
c.PetName, c.CurrSpeed);
}
}
}
```

Sadly, the compiler informs you that the `Garage` class does not implement a method named `GetEnumerator()`. This method is formalized by the `IEnumerable` interface, which is found lurking within the `System.Collections` namespace. Objects that support this behavior advertise that they are able to expose contained subitems to the caller:

```
// This interface informs the caller
// that the object's subitems can be enumerated.
public interface IEnumerable
{
IEnumerator GetEnumerator();
}
```

As you can see, the `GetEnumerator()` method returns a reference to yet another interface named `System.Collections.IEnumerator`. This interface provides the infrastructure to allow the caller to traverse the internal objects contained by the `IEnumerable`-compatible container:

```
// This interface allows the caller to
// obtain a container's subitems.
public interface IEnumerator
{
bool MoveNext ();        // Advance the internal position of the cursor.
```

```
object Current { get;}     // Get the current item (read-only property).
void Reset ();           // Reset the cursor before the first member.
}
```

If you wish to update the `Garage` type to support these interfaces, you could take the long road and implement each method manually. While you are certainly free to provide customized versions of `GetEnumerator()`, `MoveNext()`, `Current`, and `Reset()`, there is a simpler way. As the `System.Array` type (as well as many other types) already implements `IEnumerable` and `IEnumerator`, you can simply delegate the request to the `System.Array` as follows:

```
using System.Collections;
...
public class Garage : IEnumerable
{
// System.Array already implements IEnumerator!
private Car[] carArray;

public Garage()
{
carArray = new Car[4];
carArray[0] = new Car("FeeFee", 200, 0);
carArray[1] = new Car("Clunker", 90, 0);
carArray[2] = new Car("Zippy", 30, 0);
carArray[3] = new Car("Fred", 30, 0);
}

public IEnumerator GetEnumerator()
{
// Return the array object's IEnumerator.
return carArray.GetEnumerator();
}
}
```

Once you have updated your `Garage` type, you can now safely use the type within the C# `foreach` construct. Furthermore, given that the `GetEnumerator()` method has been defined publicly, the object user could also interact with the `IEnumerator` type:

```
// Manually work with IEnumerator.
IEnumerator i = carLot.GetEnumerator();
i.MoveNext();
Car myCar = (Car)i.Current;
Console.WriteLine("{0} is going {1} MPH", myCar.PetName, myCar.CurrSpeed);
```

If you would prefer to hide the functionality of `IEnumerable` from the object level, simply make use of explicit interface implementation:

```
public IEnumerator IEnumerable.GetEnumerator()
{
// Return the array object's IEnumerator.
return carArray.GetEnumerator();
}
```

6.9Building Cloneable Objects (ICloneable)

As you recall from Chapter 3, `System.Object` defines a member named `MemberwiseClone()`. This method is used to obtain a *shallow copy* of the current object. Object users do not call this method directly (as it is protected); however, a given object may call this method itself during the *cloning* process. To illustrate, assume you have a class named `Point`:

```
// A class named Point.
public class Point
{
// Public for easy access.
public int x, y;
public Point(int x, int y) { this.x = x; this.y = y;}
public Point(){}

// Override Object.ToString().
public override string ToString()
{ return string.Format("X = {0}; Y = {1}", x, y ); }
}
```

Given what you already know about reference types and value types (Chapter 3), you are aware
that if you assign one reference variable to another, you have two references pointing to the same object in memory. Thus, the following assignment operation results in two references to the same `Point` object on the heap; modifications using either reference affect the same object on the heap:

```
static void Main(string[] args)
{
// Two references to same object!
Point p1 = new Point(50, 50);
Point p2 = p1;
p2.x = 0;
Console.WriteLine(p1);
Console.WriteLine(p2);
}
```

When you wish to equip your custom types to support the ability to return an identical copy of
itself to the caller, you may implement the standard `ICloneable` interface. This type defines a single method named `Clone()`:

```
public interface ICloneable
{
object Clone();
}
```

Obviously, the implementation of the `Clone()` method varies between objects. However, the basic functionality tends to be the same: Copy the values of your member variables into a new object instance, and return it to the user. To illustrate, ponder the following update to the `Point` class:

```
// The Point now supports "clone-ability."
```

```
public class Point : ICloneable
{
public int x, y;
public Point(){ }
public Point(int x, int y) { this.x = x; this.y = y;}

// Return a copy of the current object.
public object Clone()
{ return new Point(this.x, this.y); }

public override string ToString()
{ return string.Format("X = {0}; Y = {1}", x, y ); }
}
```

In this way, you can create exact stand-alone copies of the `Point` type, as illustrated by the following code:

```
static void Main(string[] args)
{
// Notice Clone() returns a generic object type.
// You must perform an explicit cast to obtain the derived type.
Point p3 = new Point(100, 100);
Point p4 = (Point)p3.Clone();

// Change p4.x (which will not change p3.x).
p4.x = 0;

// Print each object.
Console.WriteLine(p3);
Console.WriteLine(p4);
}
```

While the current implementation of `Point` fits the bill, you can streamline things just a bit.

Because the `Point` type does not contain reference type variables, you could simplify the implementation of the `Clone()` method as follows:

```
public object Clone()
{
// Copy each field of the Point member by member.
return this.MemberwiseClone();
}
```

Be aware, however, that if the `Point` did contain any reference type member variables, `MemberwiseClone()` will copy the references to those objects (aka a *shallow copy*). If you wish to support a true deep copy, you will need to create a new instance of any reference type variables during thecloning process.

6.10 Building Comparable Objects (IComparable)

The System.IComparable interface specifies a behavior that allows an object to be sorted based on
some specified key. Here is the formal definition:

```
// This interface allows an object to specify its
// relationship between other like objects.
public interface IComparable{
int CompareTo(object o);
}
```

Let's assume you have updated the Car class to maintain an internal ID number (represented by a simple integer named carID) that can be set via a constructor parameter and manipulated
using a new property named ID. Here are the relevant updates to the Car type:

```
public class Car
{
...
private int carID;
public int ID
{
get { return carID; }
set { carID = value; }
}
public Car(string name, int currSp, int id)
{
currSpeed = currSp;
petName = name;
carID = id;
}
...
}
```

Object users might create an array of Car types as follows:

```
static void Main(string[] args)
{
// Make an array of Car types.
Car[] myAutos = new Car[5];
myAutos[0] = new Car("Rusty", 80, 1);
myAutos[1] = new Car("Mary", 40, 234);
myAutos[2] = new Car("Viper", 40, 34);
myAutos[3] = new Car("Mel", 40, 4);
myAutos[4] = new Car("Chucky", 40, 5);
}
```

As you recall, the System.Array class defines a static method named Sort(). When you invoke
this method on an array of intrinsic types (int, short, string, etc.), you are able to sort the items in the array in numerical/alphabetic order as these intrinsic data types implement IComparable. However, what if you were to send an array of Car types into the Sort() method as follows?

```
// Sort my cars?
Array.Sort(myAutos);
```

If you run this test, you would find that an `ArgumentException` exception is thrown by the runtime,
with the following message: "At least one object must implement IComparable." When you build custom types, you can implement `IComparable` to allow arrays of your types to be sorted. When you flesh out the details of `CompareTo()`, it will be up to you to decide what the baseline of the ordering operation will be. For the `Car` type, the internal `carID` seems to be the most logical candidate:

```
// The iteration of the Car can be ordered
// based on the CarID.
public class Car : IComparable
{
...
// IComparable implementation.
int IComparable.CompareTo(object obj){
Car temp = (Car)obj;
if(this.carID > temp.carID)
return 1;
if(this.carID < temp.carID)
return -1;
else
return 0;
}
}
```

6.11 Exploring The Interfaces of the System.Collections Namespace

The most primitive container construct would have to be our good friend `System.Array`. As you have already seen in Chapter 3, this class provides a number of services (e.g., reversing, sorting, clearing, and enumerating). However, the simple `Array` class has a number of limitations, most notably it does not dynamically resize itself as you add or clear items. When you need to contain types in a more flexible container, you may wish to leverage the types defined within the `System.Collections` namespace (or as discussed in Chapter 10, the `System.Collections.Generic` namespace). The `System.Collections` namespace defines a number of interfaces (some of which you have already implemented during the course of this chapter). As you can guess, a majority of the collection classes implement these interfaces to provide access to their contents. Table 7-2 gives a breakdown of the core collection-centric interfaces.

Table 7-2. *Interfaces of* `System.Collections`

`System.Collections` **Interface**	**Meaning in Life**
`ICollection`	Defines generic characteristics (e.g., count and thread safety) for a collection type.

IEqualityComparer	Defines methods to support the comparison of objects for equality.
IDictionary	Allows an object to represent its contents using name/value pairs.
IDictionaryEnumerator	Enumerates the contents of a type supporting IDictionary.
IEnumerable	Returns the IEnumerator interface for a given object.
IEnumerator	Generally supports foreach-style iteration of subtypes.
IHashCodeProvider	Returns the hash code for the implementing type using a customized hash algorithm.
IKeyComparer	(This interface is new to .NET 2.0.) Combines the functionality of IComparer and IHashCodeProvider to allow objects to be compared in a "hash-code-compatible manner" (e.g., if the objects are indeed equal, they must also return the same hash code value).
IList	Provides behavior to add, remove, and index items in a list of objects. Also, this interface defines members to determine whether the implementing collection type is read-only and/or a fixed-size container.

6.11 Building a Custom container Retrofitting the Car Type with Delegates

Clearly, the previous SimpleDelegate example was intended to be purely illustrative in nature, given that there would be no reason to build a delegate simply to add two numbers. Hopefully, however, this example demystifies the process of working with delegate types. To provide a more realistic use of delegate types, let's retrofit our Car type to send the Exploded and AboutToBlow notifications using .NET delegates rather than a custom callback interface. Beyond no longer implementing IEngineEvents, here are the steps we will need to take:

- Define the AboutToBlow and Exploded delegates.
- Declare member variables of each delegate type in the Car class.
- Create helper functions on the Car that allow the caller to specify the methods maintained by the delegate member variables.
- Update the Accelerate() method to invoke the delegate's invocation list under the correct circumstances. Ponder the following updated Car class, which addresses the first three points:

```
public class Car
{
// Define the delegate types.
public delegate void AboutToBlow(string msg);
public delegate void Exploded (string msg);

// Define member variables of each delegate type.
private AboutToBlow almostDeadList;
private Exploded explodedList;

// Add members to the invocation lists using helper methods.
public void OnAboutToBlow(AboutToBlow clientMethod)
{ almostDeadList = clientMethod; }
public void OnExploded(Exploded clientMethod)
{ explodedList = clientMethod; }
...
}
```

Recommended questions :

1. What is an interface? why they are used in c# ?.
2. Write a c# program is contain interface.
3. What is a delegate? Differentiate between synchronous and asynchronous delegate, with examples.10 m
4. Write a complete c# program to calculate and display simple interest by writing appropriate methods which could be called through delegate method of programmining.
5. Which is the alternate approach to support multiple inheritance? List its major features.
6. Briefly explain, with an example,explicit interface implementation
7. Write a program in C# to accept two strings and perform the following operations?
   i) copy string2 to string 3
   ii) check string 1 ends with ”ENGG” or not. If it is true, search character ‘a’ in string 3.
   iii) insert “VTU” in sting 2 at position 6 and display it.SYSTEM

UNIT 7
Callback Interfaces, Delegates, and Events

7.1Understanding Callback Interfaces,
7.2Understanding the .NET Delegate Type,
7.3 Members of System. Multicast Delegate, The Simplest Possible Delegate Example, Building More a Elaborate Delegate Example
7.4 Understanding Asynchronous Delegates,
7.5 Understanding (and Using) Events
7.6 The Advances Keywords of C#, A Catalog of C# Keywords
7.7 Building a Custom Indexer, A Variation of the Cars Indexer Internal Representation of Type Indexer
7.8 Using C# Indexer from VB .NET. Overloading operators, The Internal Representation of Overloading Operators, The Internal Representations of Customs Conversion Routines

7.1 Understanding Callback Interfaces

As you have seen in the previous chapter, interfaces define a behavior that may be supported by various types in your system. Beyond using interfaces to establish polymorphism, interfaces may also be used as a *callback mechanism*. This technique enables objects to engage in a two-way conversation using a common set of members. To illustrate the use of callback interfaces, let's update the now familiar `Car` type in such a way that it is able to inform the caller when it is about to explode (the current speed is 10 miles below the maximum speed) and has exploded (the current speed is at or above the maximum speed). The ability to send and receive these events will be facilitated with a custom interface named `IEngineEvents`:

```
// The callback interface.
public interface IEngineEvents
{
void AboutToBlow(string msg);
void Exploded(string msg);
}
```

Event interfaces are not typically implemented directly by the object directly interested in receiving the events, but rather by a helper object called a *sink object*. The sender of the events (the `Car` type in this case) will make calls on the sink under the appropriate circumstances. Assume the sink class is called `CarEventSink`, which simply prints out the incoming messages to the console. Beyond this point, our sink will also maintain a string that identifies its friendly name:

```
// Car event sink.
public class CarEventSink : IEngineEvents
{
private string name;
public CarEventSink(){}
public CarEventSink(string sinkName)
{ name = sinkName; }

public void AboutToBlow(string msg)
{ Console.WriteLine("{0} reporting: {1}", name, msg); }
public void Exploded(string msg)
{ Console.WriteLine("{0} reporting: {1}", name, msg); }
}
```

Now that you have a sink object that implements the event interface, your next task is to pass a reference to this sink into the `Car` type. The `Car` holds onto the reference and makes calls back on the sink when appropriate. In order to allow the `Car` to obtain a reference to the sink, we will need to add a public helper member to the `Car` type that we will call `Advise()`. Likewise, if the caller wishes to detach from the event source, it may call another helper method on the `Car` type named `Unadvise()`. Finally, in order to allow the caller to register multiple event sinks (for the purposes

of multicasting), the Car now maintains an ArrayList to represent each outstanding connection:

```
// This Car and caller can now communicate
// using the IEngineEvents interface.
public class Car
{
// The set of connected sinks.
ArrayList clientSinks = new ArrayList();

// Attach or disconnect from the source of events.
public void Advise(IEngineEvents sink)
{  clientSinks.Add(sink); }

public void Unadvise(IEngineEvents sink)
{  clientSinks.Remove(sink); }
...
}
```

To actually send the events, let's update the Car.Accelerate() method to iterate over the list of connections maintained by the ArrayList and fire the correct notification when appropriate (note the Car class now maintains a Boolean member variable named carIsDead to represent the engine's state):

```
// Interface-based event protocol!
class Car
{
...
// Is the car alive or dead?
bool carIsDead;

public void Accelerate(int delta)
{
// If the car is 'dead', send Exploded event to each sink.
if(carIsDead)
{
foreach(IEngineEvents e in clientSinks)
e.Exploded("Sorry, this car is dead...");
}
else
{
currSpeed += delta;

// Send AboutToBlow event.
if(10 == maxSpeed - currSpeed)
{
foreach(IEngineEvents e in clientSinks)
e.AboutToBlow("Careful buddy!  Gonna blow!");
}
```

```
if(currSpeed >= maxSpeed)
carIsDead = true;
else
Console.WriteLine("\tCurrSpeed = {0} ", currSpeed);
}
}
```

Here is some client-side code, now making use of a callback interface to listen to the `Car` events:

```
// Make a car and listen to the events.
public class CarApp
{
static void Main(string[] args)
{
Console.WriteLine("***** Interfaces as event enablers *****");
Car c1 = new Car("SlugBug", 100, 10);

// Make sink object.
CarEventSink sink = new CarEventSink();

// Pass the Car a reference to the sink.
c1.Advise(sink);

// Speed up (this will trigger the events).
for(int i = 0; i < 10; i++)
c1.Accelerate(20);

// Detach from event source.
c1.Unadvise(sink);
Console.ReadLine();
}
}
```

7.2 Understanding the .NET Delegate Type

Before formally defining .NET delegates, let's gain a bit of perspective. Historically speaking, the Windows API makes frequent use of C-style function pointers to create entities termed *callback functions* or simply *callbacks*. Using callbacks, programmers were able to configure one function to report back to (call back) another function in the application. The problem with standard C-style callback functions is that they represent little more than a raw address in memory. Ideally, callbacks could be configured to include additional type-safe information such as the number of (and types of) parameters and the return value (if any) of the method pointed to. Sadly, this is not the case in traditional callback functions, and, as you may suspect, can therefore be a frequent source of bugs, hard crashes, and other runtime disasters. Nevertheless, callbacks are useful entities. In the .NET Framework, callbacks are still possible, and their functionality is accomplished in a much safer and more object oriented

manner using *delegates*. In essence, a delegate is a type-safe object that points to another method (or possibly multiple methods) in the application, which can be invoked at a later time. Specifically speaking, a delegate type maintains three important pieces of information:

- The *name* of the method on which it makes calls
- The *arguments* (if any) of this method
- The *return value* (if any) of this method

Defining a Delegate in C#

When you want to create a delegate in C#, you make use of the `delegate` keyword. The name of your delegate can be whatever you desire. However, you must define the delegate to match the signature of the method it will point to. For example, assume you wish to build a delegate named `BinaryOp` that can point to any method that returns an integer and takes two integers as input parameters:

```
// This delegate can point to any method,
// taking two integers and returning an
// integer.
public delegate int BinaryOp(int x, int y);
```

When the C# compiler processes delegate types, it automatically generates a sealed class deriving from `System.MulticastDelegate`. This class (in conjunction with its base class, `System.Delegate`) provides the necessary infrastructure for the delegate to hold onto the list of methods to be invoked at a later time. For example, if you examine the `BinaryOp` delegate using `ildasm.exe`

7.3 The System.MulticastDelegate and System.Delegate Base Classes

So, when you build a type using the C# `delegate` keyword, you indirectly declare a class type that derives from `System.MulticastDelegate`. This class provides descendents with access to a list that contains the addresses of the methods maintained by the delegate type, as well as several additional methods (and a few overloaded operators) to interact with the invocation list. Here are some select members of `System.MulticastDelegate`:

```
[Serializable]
public abstract class MulticastDelegate : Delegate
{
// Methods
public sealed override Delegate[] GetInvocationList();

// Overloaded operators
public static bool operator ==(MulticastDelegate d1, MulticastDelegate d2);
public static bool operator !=(MulticastDelegate d1, MulticastDelegate d2);

// Fields
private IntPtr _invocationCount;
private object _invocationList;
}
```

`System.MulticastDelegate` obtains additional functionality from its parent class, `System.Delegate`. Here is a partial snapshot of the class definition:

```
[Serializable, ClassInterface(ClassInterfaceType.AutoDual)]
public abstract class Delegate : ICloneable, ISerializable
{
// Methods
public static Delegate Combine(params Delegate[] delegates);
public static Delegate Combine(Delegate a, Delegate b);
public virtual Delegate[] GetInvocationList();
public static Delegate Remove(Delegate source, Delegate value);
public static Delegate RemoveAll(Delegate source, Delegate value);

// Overloaded operators
public static bool operator ==(Delegate d1, Delegate d2);
public static bool operator !=( Delegate d1, Delegate d2);

// Properties
public MethodInfo Method { get; }
public object Target { get; }
}
```

Now, remember that you will never directly derive from these base classes and can typically concern yourself only with the members documented in Table 8-1.

Table 8-1. *Select Members of* `System.MultcastDelegate/System.Delegate`

Inherited Member	Meaning in Life
`Method`	This property returns a `System.Reflection.MethodInfo` type that represents details of a static method that is maintained by the delegate.
`Target`	If the method to be called is defined at the object level (rather than a static method), `Target` returns an object that represents the method maintained by the delegate. If the value returned from `Target` equals null, the method to be called is a static member.
`Combine()`	In C#, you trigger this method using the overloaded `+=` operator as a shorthand notation.
`GetInvocationList()`	This method returns an array of `System.Delegate` types, each representing a particular method that may be invoked.

`Remove()` `RemoveAll()`	These static methods removes a method (or all methods) from the invocation list. In C#, the `Remove()` method can be called indirectly using the overloaded -= operator.

The Simplest Possible Delegate Example

Delegates can tend to cause a great deal of confusion when encountered for the first time. Thus, to get the ball rolling, let's take a look at a very simple example that leverages our `BinaryOp` delegate type. Here is the complete code, with analysis to follow:

```
namespace SimpleDelegate
{
// This delegate can point to any method,
// taking two integers and returning an
// integer.
public delegate int BinaryOp(int x, int y);

// This class contains methods BinaryOp will
// point to.
public class SimpleMath
{
public static int Add(int x, int y)
{ return x + y; }
public static int Subtract(int x, int y)
{ return x - y; }
}

class Program
{
static void Main(string[] args)
{
Console.WriteLine("***** Simple Delegate Example *****\n");

// Create a BinaryOp object that
// "points to" SimpleMath.Add().
BinaryOp b = new BinaryOp(SimpleMath.Add);
// Invoke Add() method using delegate.
Console.WriteLine("10 + 10 is {0}", b(10, 10));
Console.ReadLine();
}
}
}
```

Again notice the format of the `BinaryOp` delegate, which can point to any method taking two integers and returning an integer. Given this, we have created a class named `SimpleMath`, which defines two static methods that (surprise, surprise) match the pattern

defined by the `BinaryOp` delegate. When you want to insert the target method to a given delegate, simply pass in the name of the method to the delegate's constructor. At this point, you are able to invoke the member pointed to using a syntax that looks like a direct function invocation:

```
// Invoke() is really called here!
Console.WriteLine("10 + 10 is {0}", b(10, 10));
```

Under the hood, the runtime actually calls the compiler-generated `Invoke()` method. You can verify this fact for yourself if you open your assembly in `ildasm.exe` and investigate the CIL code within the `Main()` method:

```
.method private hidebysig static void Main(string[] args) cil managed
{
...
.locals init ([0] class SimpleDelegate.BinaryOp b)
ldftn int32 SimpleDelegate.SimpleMath::Add(int32, int32)
...
newobj instance void SimpleDelegate.BinaryOp::.ctor(object, native int)stloc.0
ldstr "10 + 10 is {0}"
ldloc.0
ldc.i4.s 10
ldc.i4.s 10
callvirt instance int32 SimpleDelegate.BinaryOp::Invoke(int32, int32)
...
}
```

Recall that .NET delegates (unlike C-style function pointers) are *type safe*. Therefore, if you attempt to pass a delegate a method that does not "match the pattern," you receive a compile-time error. To illustrate, assume the `SimpleMath` class defines an additional method named `SquareNumber()`:

```
public class SimpleMath
{
...
public static int SquareNumber(int a)
{ return a * a; }
}
```

Given that the `BinaryOp` delegate can *only* point to methods that take two integers and return an integer, the following code is illegal and will not compile:

```
// Error! Method does not match delegate pattern!
BinaryOp b = new BinaryOp(SimpleMath.SquareNumber);
```

A More Elaborate Delegate Example

To illustrate a more advanced use of delegates, let's begin by updating the `Car` class to include two new Boolean member variables. The first is used to determine whether your automobile is due for a wash (`isDirty`); the other represents whether the car in question is in need of a tire rotation (`shouldRotate`). To enable the object user to interact with this new state data, `Car` also defines some additional properties and an updated constructor. Here is the story so far:

```
...
```

```
// Are we in need of a wash? Need to rotate tires?
private bool isDirty;
private bool shouldRotate;

// Extra params to set bools.
public Car(string name, int max, int curr,
bool washCar, bool rotateTires)
{
...
isDirty = washCar;
shouldRotate = rotateTires;
}
public bool Dirty
{
get{ return isDirty; }
set{ isDirty = value; }
}
public bool Rotate
{
get{ return shouldRotate; }
set{ shouldRotate = value; }
}
}
```

Now, also assume the `Car` type nests a new delegate, `CarDelegate`:

```
// Car defines yet another delegate.
public class Car
{
...
// Can call any method taking a Car as
// a parameter and returning nothing.
public delegate void CarDelegate(Car c);
...
}
```

Here, you have created a delegate named `CarDelegate`. The `CarDelegate` type represents "some function" taking a `Car` as a parameter and returning void.

7.5 Understanding C# Events

Delegates are fairly interesting constructs in that they enable two objects in memory to engage in a two-way conversation. As you may agree, however, working with delegates in the raw does entail a good amount of boilerplate code (defining the delegate, declaring necessary member variables, and creating custom registration/unregistration methods). Because the ability for one object to call back to another object is such a helpful construct, C# provides the `event` keyword to lessen the burden of using delegates in the raw. When the compiler processes the `event` keyword, you are automatically provided with registration and unregistration methods as well as any necessary member variable for your delegate types. In this light, the `event` keyword is little more than syntactic sugar, which can be used to save you some typing time. Defining an event is a two-step process. First, you need to

define a delegate that contains the methods to be called when the event is fired. Next, you declare the events (using the C# `event` keyword) in terms of the related delegate. In a nutshell, defining a type that can send events entails the following pattern (shown in pseudo-code):

```
public class SenderOfEvents
{
public delegate retval AssociatedDelegate(args);
public event AssociatedDelegate NameOfEvent;
...
}
```

The events of the `Car` type will take the same name as the previous delegates (`AboutToBlow` and `Exploded`). The new delegate to which the events are associated will be called `CarEventHandler`. Here are the initial updates to the `Car` type:

```
public class Car
{
// This delegate works in conjunction with the
// Car's events.
public delegate void CarEventHandler(string msg);

// This car can send these events.
public event CarEventHandler Exploded;
public event CarEventHandler AboutToBlow;
...
}
```

Sending an event to the caller is as simple as specifying the event by name as well as any required parameters as defined by the associated delegate. To ensure that the caller has indeed registered with event, you will want to check the event against a null value before invoking the delegate's method set. These things being said, here is the new iteration of the `Car`'s `Accelerate()` method: `public void Accelerate(int delta)`

```
{
// If the car is dead, fire Exploded event.
if (carIsDead)
{
if (Exploded != null)
Exploded("Sorry, this car is dead...");
}

else
{
currSpeed += delta;

// Almost dead?
if (10 == maxSpeed - currSpeed
&& AboutToBlow != null)
{
AboutToBlow("Careful buddy! Gonna blow!");
}
```

```
// Still OK!
if (currSpeed >= maxSpeed)
carIsDead = true;
else
Console.WriteLine("->CurrSpeed = {0}", currSpeed);
}
}
```

With this, you have configured the car to send two custom events without the need to define custom registration functions. You will see the usage of this new automobile in just a moment, but first, let's check the event architecture in a bit more detail.

7.6 The Advanced Keywords of C#,A Catalog of C# keywords

To close this chapter, you'll examine some of the more esoteric C# keywords:

- `checked/unchecked`
- `unsafe/stackalloc/fixed/sizeof`

To start, let's check out how C# provides automatic detection of arithmetic overflow and underflow conditions using the `checked` and `unchecked` keywords.

The checked Keyword

As you are no doubt well aware, each numerical data type has a fixed upper and lower limit (which may be obtained programmatically using the `MaxValue` and `MinValue` properties). Now, when you are performing arithmetic operations on a specific type, it is very possible that you may accidentally *over- flow* the maximum storage of the type (i.e., assign a value that is greater than the maximum value) or *underflow* the minimum storage of the type (i.e., assign a value that is less than the minimum value). To keep in step with the CLR, I will refer to both of these possibilities collectively as "overflow." (As you will see, checked overflow and underflow conditions result in `System.OverflowException` type. There is no `System.UnderflowException` type in the base class libraries.) To illustrate the issue, assume you have created two `System.Byte` types (a C# byte), each of which has been assigned a value that is safely below the maximum (255). If you were to add the values of these types (casting the resulting integer as a byte), you would assume that the result would be the exact sum of each member:

```
namespace CheckedUnchecked
{
class Program
{
static void Main(string[] args)
{
// Overflow the max value of a System.Byte.
Console.WriteLine("Max value of byte is {0}.", byte.MaxValue);
Console.WriteLine("Min value of byte is {0}.", byte.MinValue);
byte b1 = 100;
byte b2 = 250;
byte sum = (byte)(b1 + b2);
```

```
// sum should hold the value 350, however...
Console.WriteLine("sum = {0}", sum);
Console.ReadLine();
}
}
}
```

If you were to view the output of this application, you might be surprised to find that sum contains the value 94 (rather than the expected 350). The reason is simple. Given that a `System.Byte` can hold a value only between 0 and 255 (inclusive, for a grand total of 256 slots), `sum` now contains the overflow value (350 – 256 = 94). As you have just seen, if you take no corrective course of action, overflow occurs without exception. At times, this hidden overflow may cause no harm whatsoever in your project. Other times, this loss of data is completely unacceptable. To handle overflow or underflow conditions in your application, you have two options. Your first choice is to leverage your wits and programming skills to handle all overflow conditions manually. Assuming you were indeed able to find each overflow condition in your program, you could resolve the previous overflow error as follows:

```
// Store sum in an integer to prevent overflow.
byte b1 = 100;
byte b2 = 250;
int sum = b1 + b2;
```

Of course, the problem with this technique is the simple fact that you *are* human, and even your best attempts may result in errors that have escaped your eyes. Given this, C# provides the `checked` keyword. When you wrap a statement (or a block of statements) within the scope of the `checked` key- word, the C# compiler emits specific CIL instructions that test for overflow conditions that may result when adding, multiplying, subtracting, or dividing two numerical data types. If an overflow has occurred, the runtime will throw a `System.OverflowException` type. Observe the following update:

```
class Program
{
static void Main(string[] args)
{
// Overflow the max value of a System.Byte.
Console.WriteLine("Max value of byte is {0}.", byte.MaxValue);
byte b1 = 100;
byte b2 = 250;

try
{
byte sum = checked((byte)(b1 + b2));
Console.WriteLine("sum = {0}", sum);
}
catch(OverflowException e)
{  Console.WriteLine(e.Message); }
}
```

```
}
```

Here, you wrap the addition of `b1` and `b2` within the scope of the `checked` keyword. If you wish to force overflow checking to occur over a block of code, you can interact with the `checked` keyword as follows:

```
try
{
checked
{
byte sum = (byte)(b1 + b2);
Console.WriteLine("sum = {0}", sum);
}
}
catch(OverflowException e)
{
Console.WriteLine(e.Message);
}
```

In either case, the code in question will be evaluated for possible overflow conditions automatically, which will trigger an overflow exception if encountered.

Setting Projectwide Overflow Checking

Now, if you are creating an application that should never allow silent overflow to occur, you may find yourself in the annoying position of wrapping numerous lines of code within the scope of the `checked` keyword. As an alternative, the C# compiler supports the `/checked` flag. When enabled, *all* of your arithmetic will be evaluated for overflow without the need to make use of the C# `checked` keyword. If overflow has been discovered, you will still receive a runtime `OverflowException`. To enable this flag using Visual Studio 2005, open your project's property page and click the Advanced button on the Build tab. From the resulting dialog box, select the "Check for arithmetic overflow/underflow" check box.

7.7 Building a Custom Indexer

As programmers, we are very familiar with the process of accessing discrete items contained within a standard array using the index operator, for example:

```
// Declare an array of integers.
int[] myInts = { 10, 9, 100, 432, 9874};

// Use the [] operator to access each element.
for(int j = 0; j < myInts.Length; j++)
Console.WriteLine("Index {0} = {1} ", j, myInts[j]);
```

The previous code is by no means a major newsflash. However, the C# language provides the capability to build custom classes and structures that may be indexed just like a standard array. It should be no big surprise that the method that provides the capability to access items in this manner is termed an *indexer*.

Before exploring how to create such a construct, let's begin by seeing one in action. Assume you have added support for an indexer method to the custom collection (`Garage`) developed in

```
// Indexers allow you to access items in an arraylike fashion.
public class Program
{
static void Main(string[] args)
{
Console.WriteLine("***** Fun with Indexers *****\n");

// Assume the Garage type has an indexer method.
Garage carLot = new Garage();

// Add some cars to the garage using indexer.
carLot[0] = new Car("FeeFee", 200);
carLot[1] = new Car("Clunker", 90);
carLot[2] = new Car("Zippy", 30);

// Now obtain and display each item using indexer.
for (int i = 0; i < 3; i++)
{
Console.WriteLine("Car number: {0}", i);
Console.WriteLine("Name: {0}", carLot[i].PetName);
Console.WriteLine("Max speed: {0}", carLot[i].CurrSpeed);
Console.WriteLine();
}
Console.ReadLine();
}
}
```

As you can see, indexers behave much like a custom collection supporting the `IEnumerator` and `IEnumerable` interfaces. The only major difference is that rather than accessing the contents using interface types, you are able to manipulate the internal collection of automobiles just like a standard array. Now for the big question: How do you configure the `Garage` class (or any class/structure) to sup- port this functionality? An indexer is represented as a slightly mangled C# property. In its simplest form, an indexer is created using the `this[]` syntax. Here is the relevant update to the `Garage` type:

```
// Add the indexer to the existing class definition.
public class Garage : IEnumerable // foreach iteration
{
...
// Use ArrayList to contain the Car types.
private ArrayList carArray = new ArrayList();

// The indexer returns a Car based on a numerical index.
public Car this[int pos]
{
// Note ArrayList has an indexer as well!
get { return (Car)carArray[pos]; }
```

```
set { carArray[pos] = value }
}
}
```

Beyond the use of the `this` keyword, the indexer looks just like any other C# property declaration. Do be aware that indexers do not provide any array-like functionality beyond the use of the subscript operator. In other words, the object user cannot write code such as the following:

```
// Use ArrayList.Count property? Nope!
Console.WriteLine("Cars in stock: {0} ", carLot.Count);
```

To support this functionality, you would need to add your own `Count` property to the `Garage` type, and delegate accordingly:

```
public class Garage: IEnumerable
{
...
// Containment/delegation in action once again.
public int Count { get { return carArray.Count; } }
}
```

As you can gather, indexers are yet another form of syntactic sugar, given that this functionality can also be achieved using "normal" public methods. For example, if the `Garage` type did not support an indexer, you would be able to allow the outside world to interact with the internal array list using a named property or traditional accessor/mutator methods. Nevertheless, when you support indexers on your custom collection types, they integrate well into the fabric of the .NET base class libraries.

Recommended questions

1. What are delegates? Explain the members of system.MulticastDelegates. Write a program in c# to implement operator over loading of + and – for adding subtracting two square matrices.
2. Explain the two conceptual views of .Net assembly with a neat diagram. What are the core benefits of this?
3. With an example,Discuss advanced keywords of C #:checked, unchecked, un safe, stackalloc, volatile and size of.
4. Write a program in C# to sort and reverse an array of five elements using sort() and reverse() methods.
5. What do you understand by events and delegates in C #?.Give examples

UNIT -8
Understanding the Format of a .NET Assembly

8.1 Problems with Classic COM Binaries,
8.2 An Overview of .NET Assembly,
8.3 Building a Simple File Test Assembly
8.4 A C# Client Application,
8.5A Visual Basic .NET Client Application, Exploring the Car Library's Manifest
8.6 Exploring the Car Library's Types

8.1 Problems with Classic COM Binaries

.NET applications are constructed by piecing together any number of *assemblies*. Simply put, an assembly is a versioned, self-describing binary file hosted by the CLR. Now, despite the fact that .NET assemblies have exactly the same file extensions (`*.exe` or `*.dll`) as previous Win32 binaries (including legacy COM servers), they have very little in common under the hood. Thus, to set the stage for the information to come, let's ponder some of the benefits provided by the assembly format.

Assemblies Promote Code Reuse

As you have been building your console applications over the previous chapters,it may have seemed that *all* of the applications' functionality was contained within the executable assembly you were constructing. In reality, your applications were leveraging numerous types contained within the always accessible .NET code library, `mscorlib.dll` (recall that the C# compiler references `mscorlib.dll` automatically), as well as `System.Windows.Forms.dll`. As you may know, a *code library* (also termed a *class library*) is a `*.dll` that contains types intended to be used by external applications. When you are creating executable assemblies, you will no doubt be leveraging numerous system-supplied and custom code libraries as you create the application at hand. Do be aware, however, that a code library need not take a `*.dll` file extension. It is perfectly possible for an executable assembly to make use of types defined within an external executable file. In this light, a referenced `*.exe` can also be considered a "code library."

8.2 Understanding the Format of a .NET Assembly

Now that you've learned about several benefits provided by the .NET assembly, let's shift gears and get a better idea of how an assembly is composed under the hood. Structurally speaking, a .NET assembly (`*.dll` or `*.exe`) consists of the following elements:

- A Win32 file header
- A CLR file header
- CIL code
- Type metadata
- An assembly manifest
- Optional embedded resources

While the first two elements (the Win32 and CLR headers) are blocks of data that you can typically ignore, they do deserve some brief consideration. This being said, an overview of each element follows.

The Win32 File Header

The Win32 file header establishes the fact that the assembly can be loaded and manipulated by the Windows family of operating systems. This header data also identifies the kind of application (console-based, GUI-based, or `*.dll` code library) to be hosted by the Windows operating system.

8.3Building and Consuming a Single-File Assembly

To begin the process of comprehending the world of .NET assemblies, you'll first create a single-file `*.dll` assembly (named CarLibrary) that contains a small set of public types. To build a code library using Visual Studio 2005, simply select the Class Library project workspace The design of your automobile library begins with an abstract base class named `Car` that defines a number of protected data members exposed through custom properties. This class has a single abstract method named `TurboBoost()`, which makes use of a custom enumeration (`EngineState`) representing the current condition of the car's engine:

```
using System;

namespace CarLibrary
{
// Represents the state of the engine.
public enum EngineState
{ engineAlive, engineDead }

// The abstract base class in the hierarchy.
public abstract class Car
{
protected string petName;
protected short currSpeed;
protected short maxSpeed;
protected EngineState egnState = EngineState.engineAlive;

public abstract void TurboBoost();

public Car(){}
public Car(string name, short max, short curr)
{
petName = name; maxSpeed = max; currSpeed = curr;
}

public string PetName
{
get { return petName; }
set { petName = value; }
}
public short CurrSpeed
{
get { return currSpeed; }
set { currSpeed = value; }
}
public short MaxSpeed
{ get { return maxSpeed; } }
public EngineState EngineState
{ get { return egnState; } }
```

```
}
}
```

Now assume that you have two direct descendents of the `Car` type named `MiniVan` and `SportsCar`. Each overrides the abstract `TurboBoost()` method in an appropriate manner.

```
using System;
using System.Windows.Forms;

namespace CarLibrary
{
public class SportsCar : Car
{
public SportsCar(){ }
public SportsCar(string name, short max, short curr):base (name, max, curr){ }

public override void TurboBoost()
{
MessageBox.Show("Ramming speed!", "Faster is better...");
}
}
public class MiniVan : Car
{
public MiniVan(){ }
public MiniVan(string name, short max, short curr):base (name, max, curr){ }
public override void TurboBoost()
{
// Minivans have poor turbo capabilities!
egnState = EngineState.engineDead;
MessageBox.Show("Time to call AAA", "Your car is dead");
}
}
}
```

8.4 A C# Client Application

At this point you can build your client application to make use of the external types. Update your initial C# file as so:

```
using System;

// Don't forget to 'use' the CarLibrary namespace!
using CarLibrary;

namespace CSharpCarClient
{
public class CarClient
{
static void Main(string[] args)
{
// Make a sports car.
```

```
SportsCar viper = new SportsCar("Viper", 240, 40);
viper.TurboBoost();

// Make a minivan.
MiniVan mv = new MiniVan();
mv.TurboBoost();
Console.ReadLine();
}
}
}
```

This code looks just like the other applications developed thus far. The only point of interest is that the C# client application is now making use of types defined within a separate custom assembly. Go ahead and run your program. As you would expect, the execution of this program results in the display of various message boxes.

8.5 Building a Visual Basic .NET Client Application

To illustrate the language-agnostic attitude of the .NET platform, let's create another console application (VbNetCarClient), this time using Visual Basic .NET (see Figure 11-10). Once you have created the project, set a reference to `CarLibrary.dll` using the Add Reference dialog box. `Imports CarLibrary`

```
Module Module1

Sub Main()
End Sub

End Module
```

Notice that the `Main()` method is defined within a Visual Basic .NET Module type (which has nothing to do with a `*.netmodule` file for a multifile assembly). Modules are simply a Visual Basic .NET shorthand notation for defining a sealed class that can contain only static methods. To drive
this point home, here would be the same construct in C#:

```
// A VB .NET 'Module' is simply a sealed class
// containing static methods.
public sealed class Module1
{
public static void Main()
{
}
}
```

In any case, to exercise the `MiniVan` and `SportsCar` types using the syntax of Visual Basic .NET, update your `Main()` method as so:

```
Sub Main()
Console.WriteLine("***** Fun with Visual Basic .NET *****")
Dim myMiniVan As New MiniVan()
myMiniVan.TurboBoost()
```

```
Dim mySportsCar As New SportsCar()
mySportsCar.TurboBoost()
Console.ReadLine()
End Sub
```

When you compile and run your application, you will once again find a series of message boxes displayed.

8.6 Exploring the CarLibrary's Types

Recall that when you generate a strong name for an assembly, the entire public key is recorded in the assembly manifest. On a related note, when a client references a strongly named assembly, its manifest records a condensed hash value of the full public key, denoted by the `.publickeytoken` tag. If you were to open the manifest of `SharedCarLibClient.exe` using `ildasm.exe`, you would find the following:

```
.assembly extern CarLibrary
{
.publickeytoken = (21 9E F3 80 C9 34 8A 38)
.ver 1:0:0:0
}
```

If you compare the value of the public key token recorded in the client manifest with the public key token value shown in the GAC, you will find a dead-on match. Recall that a public key represents one aspect of the strongly named assembly's identity. Given this, the CLR will only load version 1.0.0.0 of an assembly named CarLibrary that has a public key that can be hashed down to the value 219EF380C9348A38. If the CLR does not find an assembly meeting this description in the GAC (and cannot find a private assembly named CarLibrary in the client's directory), a`FileNotFound` exception is thrown.

Recommended question:

1. Write short notes on the following: i) classic COM binaries versus .Net assemblies ii) Cross language inheritance.
2. Write short notes on: i)Interface of system collection ii) Indexers iii) Shared assemblies iv)Mutual and immutable strings.
3. Illustrate with an example, difference between synchronous and asynchronous delegates.