DIGITAL NOTES ON APPLICATION PROGRAMMING

B.TECH IV YEAR - I SEM
(2019-20)

DEPARTMENT OF INFORMATION TECHNOLOGY

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY
(Autonomous Institution – UGC, Govt. of India)
(Affiliated to JNTUH, Hyderabad, Approved by AICTE - Accredited by NBA & NAAC – 'A' Grade - ISO 9001:2015 Certified)
Maisammaguda, Dhulapally (Post Via. Hakimpet), Secunderabad – 500100, Telangana State, INDIA.

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

DEPARTMENT OF INFORMATION TECHNOLOGY

	L	T/P/D	C
IV Year B.Tech IT – I Sem	**5**	**1/-/-**	**4**

(R15A0570)APPLICATION PROGRAMMING

UNIT I:

MS.NET Framework Introduction: The .NET Framework - an Overview- Framework Components - Framework Versions-Types of Applications which can be developed using MS.NET - MS.NET Base Class Library - MS.NET Namespaces - MSIL / Metadata and PE files- Common Language Runtime (CLR) - Managed Code -MS.NET Memory Management / Garbage Collection -Common Type System (CTS) - Common Language Specification (CLS)- Types of JIT Compilers-Security Manager . Building C# Applications Using csc.exe, Building .NET Applications Using Notepad++, Building .NET Applications Using Visual C# 2010 Express, Building .NET Applications Using Visual Studio 2010.

UNIT II:

Developing Console Application: Introduction to Project and Solution in Studio- Entry point method - Main. - Compiling and Building Projects -Using Command Line Arguments -Importance of Exit code of an application-Different valid forms of Main-Compiling a C# program using command line utility CSC.EXE-Datatypes - Global, Stack and Heap Memory-Common Type System-Reference Type and Value Type- Data types & Variables Declaration- Implicit and Explicit Casting- Checked and Unchecked Blocks – Overflow Checks-Casting between other data types-Boxing and Unboxing-Enum and Constant-Operators-Control Statements - Working with Arrays -Working with Methods-Pass by value and by reference and out parameters

UNIT III:

Object Oriented Programming: Object -Lifecycle of an Object-relationship between Class and Object-Define Application using Objects-Principles of Object Orientation-Encapsulation –Inheritance-Polymorphism-Encapsulation is binding of State and Behavior together-understand the difference between object and reference- Working with Methods, Properties -Copy the reference in another reference variable- Abandoning the object- Constructor & Destructor-Working with "static" Members- Inheritance- Inheritance and "is a" relationship-Protected Keyword -Constructor in Inheritance -Type Casting of Reference Types- Static and Dynamic Binding and Virtual Methods-Abstract Class-Object as Parent of all classes-Namespaces-Access Specifiers-Interface & Polymorphism-Syntax for Implementation of Interface- Explicit Implementation of Interface members-Types of Inheritance-Overloading-Overriding- Partial Classes.

Exception Handling: Exception -Rules for Handling Exception - Exception classes and its important properties -Understanding & using try, catch keywords -Throwing exceptions-Importance of finally block- "using" Statement -Writing Custom Exception Classes

UNIT IV:

Delegates And Events: Understanding the .NET Delegate type, defining a Delegate Type in C#, The System. Multicast Delegate and System. Delegate Base Classes, The Simple Possible Delegate Example, Sending Object State Notification using Delegates-chat application-anonymous

PROGRAMMING WITH .NET ASSEMBLIES: Configuring .NET Assemblies, defining Custom Namespaces, The role of .NET Assemblies, Understanding the Format of a .NET assembly, Building and Consuming a Single-File Assembly, Building and Consuming a Multifile Assembly, Understanding Private Assembly, Understanding Shared Assembly, Consuming a Shared Assembly, Configuring Shared assemblies, Understanding Publisher Policy assemblies, Understanding the <codebase> Element, The System. Configuration Namespace.

UNIT V:

ADO.NET PART - I: The Connected Layer: A High-Level Definition of ADO.NET, Understanding ADO.NET Data Provider, Additional ADO.NET Namespaces, The Types of the System.Data.namespace, Abstracting Data Providers Using Interfaces, Creating the Auto Lot Database, The ADO.NET data Provider Factory Model, Understanding the Connected Layer of ADO.NET, Working with Data Readers, Building a reusable Data Access Library, Creating a Console UI-Based Front End, Understanding Database Transactions.

ADO.NET PART - II: Disconnected Layer: Understanding the Disconnected Layer of ADO.NET, Understanding the Role of the Dataset, Working with DataColumns, Working with Datarows, Working with DataTables, Binding with Data Adapters, Adding Disconnected Functionality to AutoLotDAL.dll, Multitabled Dataset Objects and Data Relationships, the Windows Forms Database Code into a Class Library, Programming with LINQ to DataSet.

TEXT BOOKS:

1.Andrew Troelsen (2010), Pro C# and the .NET 4 Platform, 5th edition, Springer (India) Private Limited, New Delhi, India.

REFERENCE BOOKS:

1.E. Balagurusamy (2004), Programming in C#, 5th edition, Tata McGraw-Hill,
New Delhi, India. 2.Herbert Schildt (2004), The Complete Reference: C#, Tata
McGraw-Hill, New Delhi, India.

3.Simon Robinson, Christian N agel, Karli Watson, Jay Gl (2006), Professional C#, 3rd edition, Wiley & Sons, India.

Websites
http://www.c-sharpcorner.com/beginners/
http://www.tutorialspoint.com/csharp/

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

DEPARTMENT OF INFORMATION TECHNOLOGY

INDEX

UNIT-I

MS.NET FRAMEWORK INTRODUCTION

INTRODUCTION

.NET Framework 2.0 Software Development Kit (SDK). Do be aware that the .NET Framework 2.0 SDK is automatically installed with Visual Studio 2005 as well as Visual C# 2005 Express;

If you are not developing with Visual Studio 2005/Visual C# 2005 Express, navigate to http://msdn.microsoft.com/netframework and search for ".NET Framework 2.0 SDK". Once you have located the appropriate page, download setup.exe and save it to a location on your hard drive. At this point, double-click the executable to install the software.

In addition to the content installed under C:\Program Files\Microsoft Visual Studio 8\SDK\v2.0,the setup program also creates the Microsoft.NET\Framework subdirectory under your Windows directory. Here you will find a subdirectory for each version of the .NET Framework installed on your machine. Within a version-specific subdirectory, you will find command-line compilers for each language that ships with the Microsoft .NET Framework (CIL, C#, Visual Basic .NET, J#, and JScript .NET), as well as additional command-line development utilities and .NET assemblies.

THE .NET SOLUTION

i. Full Interoperability with Existing Win32code

- Existing COM binaries can inter-operate with newer .NET binaries & vice versa.
- Also, PInvoke(Platform invocation) allows to invoke raw C-based functions from managed-code.

ii. Complete & Total Language Integration

- .NET supports

  → cross language inheritance

  → cross language exception-handling &

  → cross language debugging

iii. A Common Runtime Engine Shared by all .NET-aware languages

- Main feature of this engine is "a well-defined set of types that each .NET-aware language understands".

iv. A Base Class Library that

  → protects the programmer from the complexities of raw API calls

  → offers a consistent object-model used by all .NET-aware languages

v. A Truly Simplified Deployment Model

- Under .NET, there is no need to register a binary-unit into the system-registry.
- .NET runtime allows multiple versions of same *.dll to exist in harmony on a single machine.

vi. No more COM plumbing IClasFactory, IUnknown, IDispatch, IDL code and the evil VARIANT compliant data-types have no place in a native .NET binary.

C# LANGUAGE OFFERS THE FOLLOWING FEATURES

• No pointer required. C# programs typically have no need for direct pointer manipulation.
• Automatic memory management through garbage-collection. Given this, C# does not support a delete keyword.
• Formal syntactic-constructs for enumerations, structures and class properties.
• C++ like ability to overload operators for a custom type without the complexity.
• Full support for interface-based programming techniques.
• Full support for aspect-based programming techniques via attributes. This brand of development allows you to assign characteristics to types and their members to further qualify the behavior.

THE BUILDING BLOCKS OF THE .NET PLATFORM

• .NET can be understood as
→ a new runtime environment &
→ a common base class library (Figure:1-1)
• Three building blocks are:
→ CLR (Common Language Runtime)
→ CTS (Common Type System)
→ CLS (Common Language Specification)
• Runtime-layer is referred to as CLR.
• Primary role of CLR: to locate, load & manage .NET types.
• In addition, CLR also takes care of
→ automatic memory-management
→ language-integration &
→ ensuring type-safety
• CTS
→ describes all possible data-types & programming-constructs supported by runtime
→ specifies how these entities interact with each other &
→ specifies how they are represented in metadata format
• CLS define a subset of common data-types & programming-constructs that all .NET aware- languages can agree on.
• Thus, the .NET types that expose CLS-compliant features can be used by all .NET-aware languages.
• But, a data type or programming construct, which is outside the bounds of the CLS, may not be used by every .NET programming language.

The Role of Base Class Library (BCL)

• BCL is a library of functionality available to all .NET-aware languages.
• This encapsulates various primitives for
→ file reading & writing
→ threads
→ file IO
→ graphical rendering
→ database interaction
→ XML document manipulation
→ programmatic security
→ construction of web enabled front end

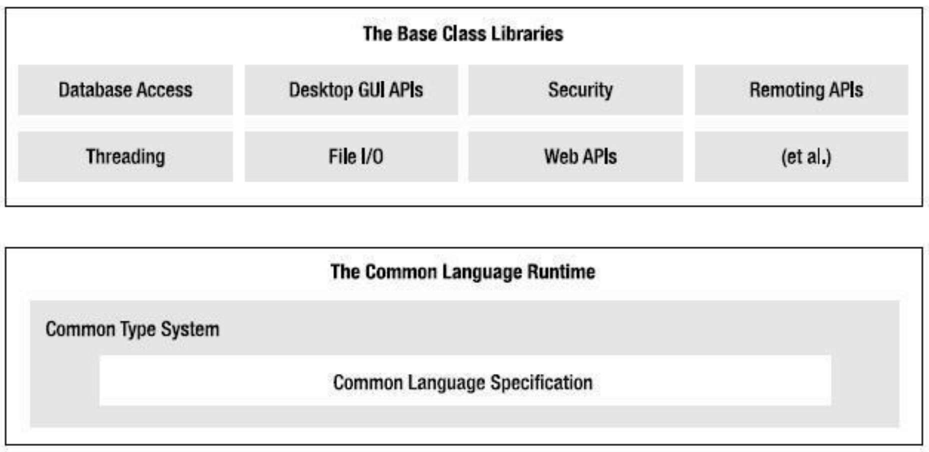


Figure 1-1. The CLR, CTS, CLS, and base class library relationship

AN OVERVIEW OF .NET BINARIES

• Regardless of which .NET-aware language you choose (like C#, VB.NET, VC++.NET etc),
→.NET binaries take same file-extension (.exe or .dll)
→.NET binaries have absolutely no internal similarities (Figure:1-2)

• .NET binaries do not contain platform-specific instruction but rather platform-agnostic "Common Intermediate Language (CIL)".

• When .NET binaries have been created using a .NET-aware compiler, the resulting module is bundled into an *assembly*.

• An assembly is a logical grouping of one or more related modules (i.e. CIL, metadata, manifest) that are intended to be deployed as a single unit.

• An assembly contains CIL-code which is not compiled to platform-specific instructions until absolutely-necessary.

• Typically "absolutely-necessary" is the point at which "a block of CIL instructions" are referenced for use by the runtime-engine.

• The assembly also contains *metadata* that describes the characteristics of every "type" living within the binary.

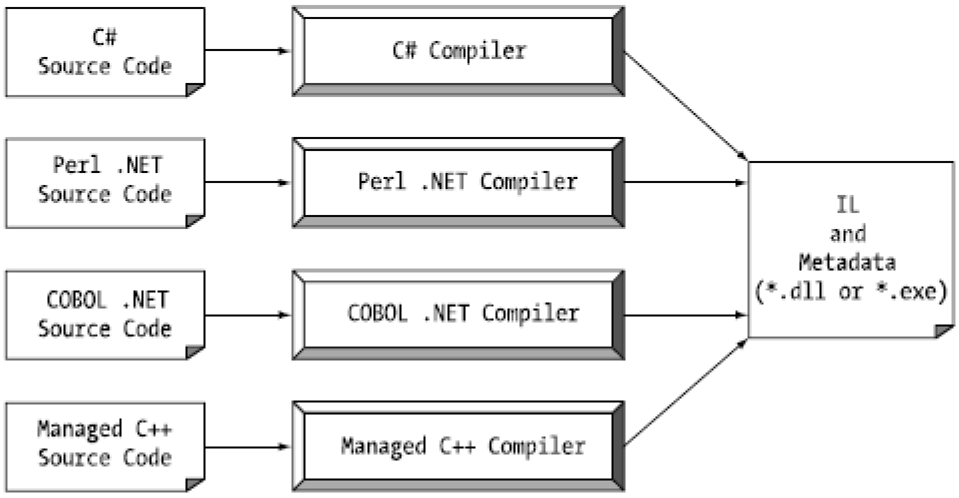
• Assemblies themselves are also described using metadata, which is termed as *manifest.*

• The manifest contains information such as
→ name of assembly/module
→ current version of assembly
→ list of files in assembly
→ copyright information
→ list of all externally referenced assemblies that are required for proper execution

Managed code & unmanaged code

• C# produces the code that can execute within the .NET runtime.The code targeting the .NET runtime is called as managed-code.

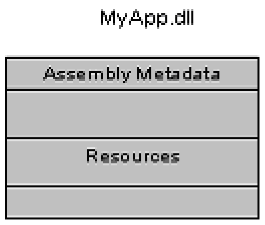
• Conversely, code that cannot be directly hosted by the .NET runtime is termed unmanaged-code.



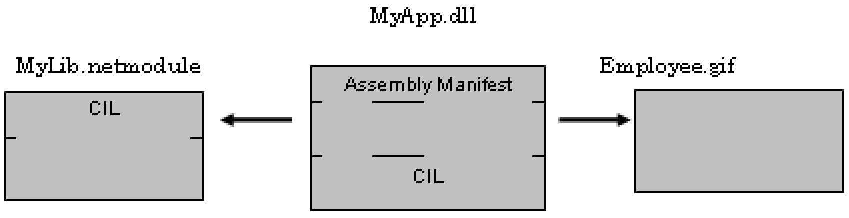
SINGLE-FILE AND MULTI-FILE ASSEMBLIES

- Single-file assemblies contain all the necessary CIL, metadata and manifest in a single well-defined package.
- On the other hand, multi-file assemblies are composed of numerous .NET binaries, each of which is termed a *module.*
- When building a multiple 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.
- The primary module maintains the set of required secondary modules within the assembly manifest.
- When building a multiple assembly, one of the module (termed as primary module) must contain the assembly-manifest. The other related modules contain a module-level manifest, CIL and type metadata.
- Multiple assemblies are used when different modules of the application are written in different languages.
- Q: Why would you choose to create a multi-file assembly?

Ans: Multiple assemblies make the downloading process more efficient. They enable you to store the seldom used types in separate modules and download them only when needed



Single-File Assembly



MultiFile Assembly

ROLE OF CIL

• CIL is a language that sits above any particular platform-specific instruction set.

• Regardless of which .NET-aware language you choose (like C#, VB.NET, VC++.NET etc), the associated compiler produces CIL instructions.

• Once the C# complier (csc.exe) compiles the 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 program.

Benefits of CIL

• *Language Integration*: Each .NET-aware language produces the same underlying-CIL. Therefore, all
.NET-aware languages are able to interact within a well-defined binary arena.

• Since CIL is platform-agnostic, the .NET runtime is poised to become a *platform-independent architecture*. Thus, .NET has the potential to allow you to develop an application in any language and have it run on any platform.

THE ROLE OF METADATA

• Metadata describes each and every type (class, structure, enumeration) defined in the binary, as well as the members of each type (properties, methods, events)

• It describes each externally referenced assembly that is required by the executing assembly to operate correctly.

• It is used
  → by numerous aspects of the .NET runtime environment
  → by various development tools
  → by various object browsing utilities, debugging tools and even the C# compiler itself

• It is the backbone of numerous .NET technologies such as .NET Remoting, reflection services and object serialization.

- Consider the following example

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

- Within the resulting "MetaInfo" window, you will find a description of the Add() method looking

something like the following:

```
Method #2
-------------------------------------------------------------------------
MethodName: Add (06000002)
RVA:      000002064
ImplFlags: [IL] [Managed] (00000000) 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)
```

- In above metadata, you can see that Add() method, return type and method arguments have been fully described by the C# compiler.

THE ROLE OF MANIFEST

- Assemblies themselves are also described using metadata, which is termed as *manifest.*
- The manifest contains information such as
  →name of assembly/ module
  →current version of the assembly
  →list of files in assembly
  →copyright information
  →list of all externally referenced assemblies that are required for proper execution

COMPILING CIL TO PLATFORM-SPECIFIC INSTRUCTIONS

- Assemblies contain CIL instructions and metadata, rather than platform-specific instructions.
- CIL must be compiled on-the-fly before use.
- Jitter(Just-in-time compiler) is used to compile the CIL into meaningful CPU instructions.
- The .NET runtime environment forces a JIT compiler for each CPU targeting the CLR, each of which is optimized for the platform it is targeting.
- Developers can write a single body of code that can be efficiently JIT-compiled and executed on machines with different architectures. 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 a 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

• Jitter will cache the results in memory in a manner suited to the target OS. For example, 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.

UNDERSTANDING THE CTS

• 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}.
• CTS
  → **fully describes all possible data-types & programming constructs supported by the runtime**
  → **specifies how these entities can interact with each other &**
  → **details of how they are represented in the metadata format**
• When you wish to build assemblies that can be used by all possible .NET-aware languages, you need to conform your exposed types to the rules of the CLS.

CTS Class Types

• A class may be composed of any number of members (methods, constructor) and data points (fields).
• CTS allow a given class to support virtual and abstract members that define a polymorphic interface for derived class.
• Classes may only derive from a single base class (multiple inheritances are not allowed for class).
• In C#, classes are declared using the class keyword. For example,

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

Class Characteristic	***Meaning***
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 feature 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

• A structure can be thought of as a lightweight type having value-semantics.
• Structure may define any number of parameterized constructors.
• All structures are derived from a common base class: *System.ValueType*.
• This base class configures a type to behave as a stack-allocated entity rather than a heap-allocated entity.
• The CTS permits structures to implement any number of interfaces; but, structures may not become a base type to any other classes or structures. Therefore structures are explicitly sealed.
• In C#, structure is declared using the struct keyword. For example,

```
// 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 Type

• Interface is a named collection of abstract member definitions, which may be supported by a given class or structure.
• Interfaces do not derive from a common base type (not even System.Object)
• When a class/structure implements a given interface, you are able to request access to the supplied functionality using an interface-reference in a polymorphic manner.
• When you create custom interface using a .NET-aware language, the CTS allows a given interface to derive from multiple base interfaces.
• In C#, interface types are defined using the interface keyword, for example:

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

CTS Enumeration Types

• Enumeration is used to group name/value pair under a specific name.
• By default, the storage used to hold each item is a System.Int32 (32-bit integer)
• Enumerated types are derived from a common base class, *System.Enum* This base class defines a number of interesting members that allow you to extract, manipulate, and transform the underlying name/value pairs programmatically.
• Consider an example of 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 as:

```
// A C# enumeration
public enum playertype
{
        wizard=10,
        fighter=20,
        thief=30
};
```

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.
- Delegates are useful when you wish to provide a way for one entity to forward a call to another entity.
- Delegates provide intrinsic support for multicasting, i.e. forwarding a request to multiple recipients.
- They also provide asynchronous method invocations.
- They provide the foundation for the .NET event architecture.
- In C#, delegates are declared using the delegate keyword as shown in the following example:

```
// 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);
```

INTRINSIC CTS DATA TYPES

.NET Base Type (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.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

UNDERSTANDING THE CLS

- The Common Language Specification (CLS) is a set of rules that describe the small and complete set of features.
- These features are supported by a .NET-aware compiler to produce a code that can be hosted by CLR.
- Also, this code can be accessed by all languages in the .NET platform.
- The CLS can be viewed as physical subset of the full functionality defined by the CTS.
- The CLS is a set of rules that compiler builders must conform to, if they intend their products to function seamlessly within the .NET-universe.
- Each rule describes how this rule affects those who build the compilers as well as those who interact with them. For example, the CLS Rule 1 says:

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

- Given this rule, we can understand 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 match 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;
        }
}
```

•We can 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; temp= x+y; return temp;
        }
}
```

•Now, we have a match to the rules of the CLS, and can assured that all .NET languages are able to invoke the Add() method.

Ensuring CLS compliance

- C# does define a number of programming constructs that are not CLS-compliant. But, we can instruct the C# compiler to check the code for CLS compliance using a single .NET attribute:

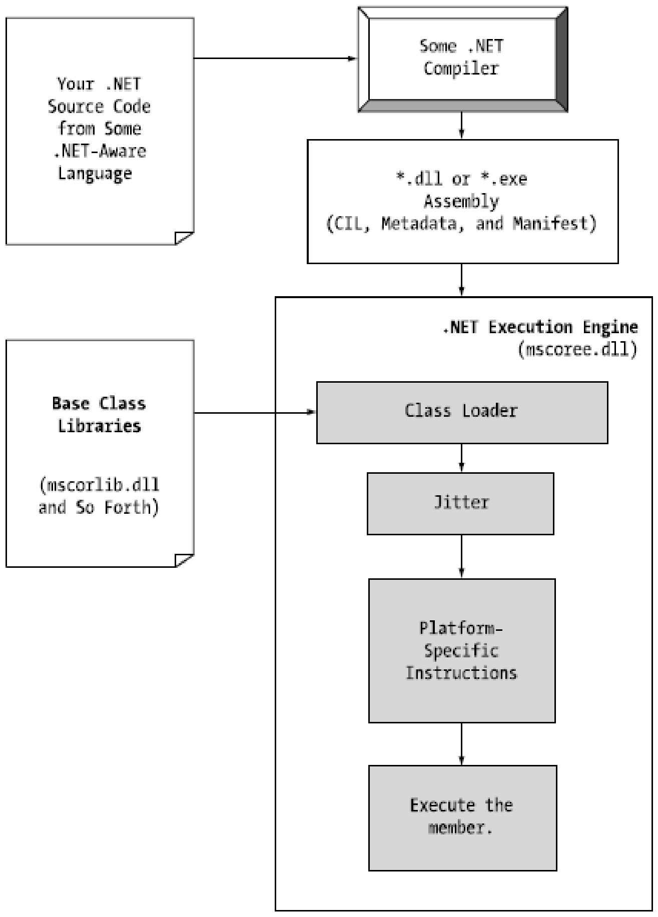
```
// Tell the C# compiler to check for CLS compliance.
```

[assembly: System.CLSCompliant(true)]

This statement must be placed outside the scope of any namespace. The [CLSCompliant] attribute will instruct the C# compiler to check each and every line of code against the rules of the CLS. If any CLS violations are discovered, we will receive a compiler error and a description of the offending code

UNDERSTANDING THE CLR

- The runtime can be understood as a collection of external services that are required to execute a given compiled unit-of-code. (Figure: 1-3)
- .NET runtime provides a single well-defined runtime layer that is shared by all .NET aware languages.
- The heart of CLR is physically represented by an assembly named *mscoree.dll* (Common Object Runtime Execution Engine)
- When an assembly is referenced for use, mscoree.dll is loaded automatically, which in turn loads the required assembly into memory.
- Runtime-engine
  - → **lays out the type in memory**
  - → **compiles the associated CIL into platform-specific instruction**
  - → **performs any security checks and**
  - → **then executes the code**
- In addition, runtime-engine is in charge of
  - → **resolving location of an assembly and**
  - → **finding requested type within binary by reading contained-metadata**
- In addition, runtime-engine will also interact with the types contained within the base class libraries.
- *mscorlib.dll* assembly contains a large number of core types that encapsulate a wide variety of common programming tasks as well as the core data types used by all .NET languages.


Your .NET Source Code from Some .NET-Aware Language
Some .NET Compiler
*.dll or *.exe Assembly (CIL, Metadata, and Manifest)
.NET Execution Engine (mscoree.dll)
Base Class Libraries
(mscorlib.dll and So Forth)
Class Loader
Jitter
Platform-Specific Instructions
Execute the member.

A TOUR OF THE .NET NAMESPACES

- C# language does not come with a pre-defined set of language-specific classes.
- There is no C# class library. Rather, C# developers use existing types supplied by the .NET framework.
- Advantage of namespaces: any language targeting the .NET runtime makes use of same namespaces and same types as a C# developer.
- Namespace is a way to group semantically related types (classes, enumerations, interfaces, delegates and structures) under a single umbrella. For example, the System.IO namespace contains file I/O related types; the System.Data namespace defines basic database types, and so on

```
//Hello world in C#

using System; public
class MyApp
{
        public static void Main()
        {
                Console.WriteLine("hi from C#");
        }
}
```

```
'Hello world in VB.NET

Imports System Public
Module MyApp
        Sub Main()
                Console.WriteLine("hi from VB.NET")
        End Sub
End Module
```

- In above Hello world code, each language is making use of the “Console” class defined in the “System” namespace.
- "System" namespace provides a core body of types that you will need to use as a .NET developers.
- You cannot build any sort of functional C# application without making a reference to the "System" namespace.
- “System" is the root namespace for numerous other .NET namespaces.

.NET NAMESPACES

System

In this, you find numerous low-level classes dealing with primitive types, mathematical manipulations, garbage collection, exceptions and predefined attributes.

System.Collections

This defines a number of stock container objects (ArrayList, Queue, etc) as well as base types and interfaces that allow you to build customized collections.

System.Data.OleDb These are used for database manipulation.

System.Diagnostics

In this, you find numerous types that can be used by any .NET-aware language to programmatically debug and trace your source code.

System.Drawing System.Drawing.Printing

In this, you find numerous types wrapping GDI+ primitives such as bitmaps, fonts, icons, printing support, and advanced graphical rendering support.

System.IO

This includes file IO and buffering.

System.Net

This contains types related to network programming (requests/responses, sockets, end points)

System.Security

In this, you find numerous types dealing with permissions, cryptography and so on.

System.Threading

This deals with threading issues. In this, you will find types such as Mutex, Thread and Timeout.

System.Web

This is specifically geared toward the development of .NET Web applications, including ASP.NET & XML Web services.

System.Windows

This facilitates the construction of more traditional main windows, dialog boxes and custom widgets.

System.Xml

This contains numerous types that represent core XML primitives and types used to interact with XML data.

Accessing a Namespace Programmatically

- In C#, the “using” keyword simplifies the process of referencing types defined in a particular namespace. In a traditional desktop application, we can include any number of namespaces like –

```
using System;                    // General base class
library types. using System.Drawing;          //
Graphical rendering types.
```

- Once we specify a namespace, we can create instances of the types they contain. For example, if we are interested in creating an instance of the Bitmap class, we can write:

```
using System;
using System.Drawing; class
MyApp
{
        public void DisplayLogo()
        {
                // create a 20x20 pixel bitmap. Bitmap bm =
                new Bitmap(20, 20);
                ...
        }
}
```

- As this application is referencing System.Drawing, the compiler is able to resolve the Bitmap class as a member of this namespace. If we do not specify the System.Drawing namespace, we will get a compiler error. However, we can declare variables using a fully qualified name as well:

```
using System;
class MyApp
{
        public void DisplayLogo()
        {
                // Using fully qualified name.
                System.Drawing.Bitmap bm =new
                System.Drawing.Bitmap(20, 20);
                ...
        }
}
```

Following Techniques can be used to learn more about the .NET Libraries

- .NET SDL online documentation
- The ildasm.exe utility
- The class viewer web application.
- The wincv.exe desktop application.
- The visual studio .NET integrated object browser.

Deploying the .NET Runtime

- The .NET assemblies can be executed only on a machine that has the .NET Framework installed.
- But, we can not copy and run a .NET application in a computer in which .NET is not installed. 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.
- 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, Global Assembly Cashe).

UNIT-II

DEVELOPING CONSOLE APPLICATION

BUILDING A C# APPLICATION USING csc.exe

- Consider the following code:

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

- Once you finished writing the code in some editor, save the file as *TestApp.cs*
- To compile & run the program, use the following command-set:

```
Output:
C:\CSharpTestApp> csc TestApp.cs
C:\CSharpTestApp> TestApp.exe
  Testing 1 2 3
```

- List of Output Options of the C# Compiler:
  1) **/out** : This is used to specify name of output-file to be created. By default, name of output-file is same as name of input-file.
  2) /**target:exe** :This builds an executable console application. This is the default file output- type.
  3) **/target:library** :This option builds a single-file assembly.
  4) **/target:module** :This option builds a module. Modules are elements of multi-file assemblies.
  5) **/target:winexe** :Although you are free to build windows-based applications using the /target:exe flag, this flag prevents an annoying console window from appearing in the background.

REFERENCING EXTERNAL ASSEMBLIES

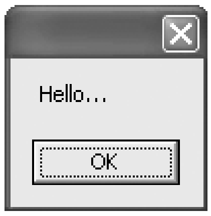
- Consider the following code:

```
using System;
using System.Windows.Forms; class
TestApp
{
        public static void Main()
        {
                MessageBox.Show("Hello . . .");
        }
}
```

- As you know, *mscorlib.dll* is automatically referenced during the compilation process. But if you want to disable this option, you can specify using /nostdlib flag.
- Along with "using" keyword, you must also inform compiler which assembly contains the referenced namespace. For this, you can use the /reference(or /r) flag as follows:

csc /r:System.Windows.Forms.dll TestApp.cs

• Now, running the application will give the output as –



COMPILING MULTIPLE SOURCE FILES

• When we have more than one source-file, we can compile them together. For illustration, consider the following codes:

File Name: HelloMessage.cs

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

File Name: TestApp.cs

```
        using
        System;
        class
        TestAp
        p
        {
                public static void Main()
                {
                        Console.WriteLine("Testing 1 2 3");
                        HelloMessage h= new
                        HelloMessage(); h.Speak();
                }
        }
```

• We can compile C# files by listing each input-file explicitly:

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

• As an alternative, you can make use of the wildcard character(*) to inform the compiler to include all *.cs files contained in the project-directory.

• In this case, you have to specify the name of the output-file(/out) to directly control the name of the resulting assembly:

*csc /r:System.Windows.Forms.dll /out:TestApp.exe *.cs*

REFERENCING MULTIPLE EXTERNAL ASSEMBLIES

• If we want to reference numerous external assemblies, then we can use a semicolon-delimited list. For example:

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

WORKING WITH csc.exe RESPONSE FILES

- When we are building complex C# applications, we may need to use several flags that specify numerous referenced assemblies and input-files.
- To reduce the burden of typing those flags every time, the C# compiler provides response-files.
- Response-files contain all the instructions to be used during the compilation of a program.
- By convention, these files end in a *.rsp extension.
- Consider you have created a response-file named "TestApp.rsp" that contains the following arguments:

```
# this is the response file for the TestApp.exe application # external
assembly reference
/r: System.Windows.Forms.dll
#output and files to compile using wildcard syntax
/target:exe /out:TestApp.exe *.cs
```

- Save this response-file in the same directory as the source files to be compiled. Now, you can build

your entire application as follows:

csc @TestApp.rsp

- Any flags listed explicitly on the command-line will be overridden by the options in a given response-file. Thus, if we use the statement,

    csc /out:Foo.exe @TestApp.rsp

- The name of the assembly will still be TestApp.exe(rather than Foo.exe),given the /out:TestApp.exe flag listed in the TestApp.rsp response-file.

THE DEFAULT RESPONSE FILE (csc.rsp)

- 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 we open this file using Notepad, we can see that numerous .NET assemblies have already been specified using the /r: flag.
- When we are building our C# programs using csc.exe, this file will be automatically referenced, even if we provide a custom *.rsp file.
- Because of default response-file, the current Test.exe application could be successfully compiled using the following command set

    *csc /out:Test.exe *.cs*

- If we wish to disable the automatic reading of csc.rsp, we can specify the /noconfig option:

    csc @Test.rsp /noconfig

GENERATING BUG REPORTS

- /bugreport flag allows you to specify a file that will be populated with any errors encountered during the compilation-process. The syntax for using this flag is–

    *csc /bugreport: bugs.txt *.cs*

- We can enter any corrective information for possible errors in the program, which will be saved to the specified file (i.e. bugs.txt).
- Consider the following code with an error (bug):

```
public static void Main()
{
        HelloMessage h=new HelloMessage();
        h.Speak()       // error occurs because ; is missing
}
```

• When we compile this file using /bugreport flag, the error message will be displayed and corrective
action is expected as shown –

Test.cs (23, 11): Error CS1002: ; expected

Please describe the compiler problem: _

• Now if we enter the statement like

"FORGOT TO TYPE SEMICOLON"

then, the same will be stored in the „bugs.txt‟ file.

C# "PREPROCESSOR" DIRECTIVES

• These are processed as part of the lexical analysis phase of the compiler.
• Like C, C# supports the use of various symbols that allow you to interact with the compilation process.
• List of pre-processor directives

#define, #undef
These are used to define and un-define conditional compilation symbols.
#if,#elif, #else, #endif
These are used to conditionally skip sections of source code.
#line
This is used to control the line numbers emitted for errors and warnings.
#error, #warning
These are used to issue errors and warnings for the current build**.**
#region, #endregion
These are used to explicitly mark sections of source-code.
Regions may be expanded and collapsed within the code window; other IDEs will ignore these symbols.

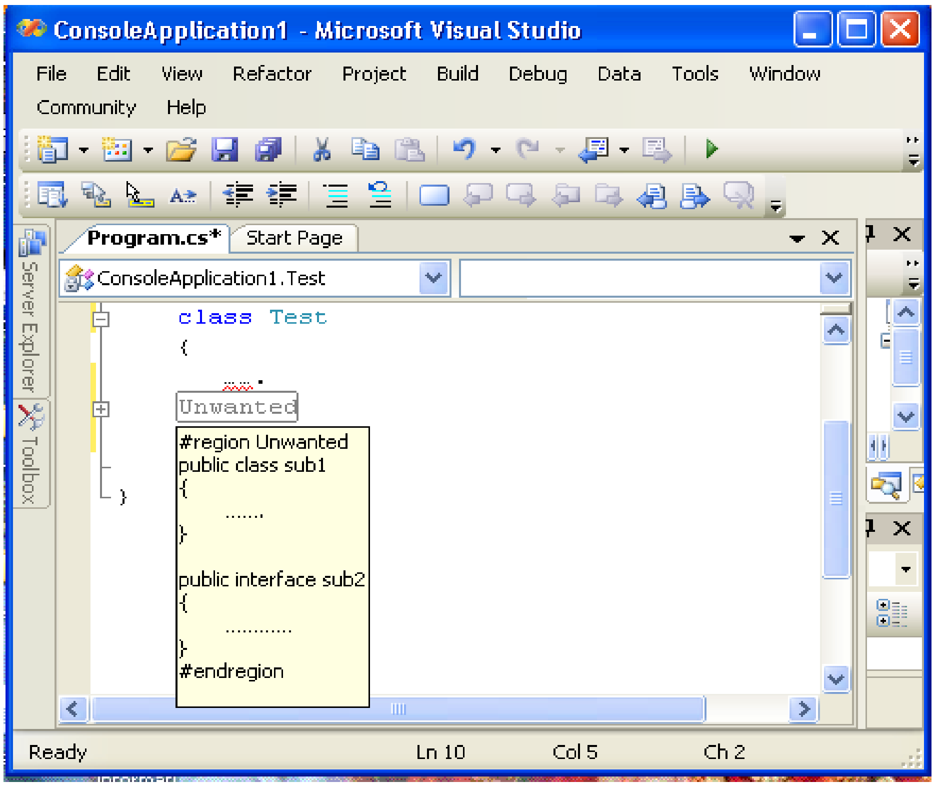
SPECIFYING CODE REGIONS

• Using #region and #endregion tags, we can specify a block of code that may be hidden from view and identified by a textual marker.
• The use of regions helps to keep lengthy *.cs files more manageable.
• For example, we can create a region for defining a type‟s constructor (may be class, structure etc), type‟s properties and so on.
• Consider the following code:

```
class Test
{
        …….
        #region Unwanted
        public class sub1
        {
                …….
        }

        public interface sub2
        {
                …………
        }
        #endregion
}
```

• Now, when we put mouse curser on that region, it will be shown as –


ConsoleApplication1 - Microsoft Visual Studio
File Edit View Refactor Project Build Debug Data Tools Window
Community Help
Program.cs* Start Page
ConsoleApplication1.Test
Server Explorer
Toolbox
Ready Ln 10 Col 5 Ch 2

```
class Test
{
    Unwanted
}
```

```
#region Unwanted
public class sub1
{
        .......
}

public interface sub2
{
        ............
}
#endregion
```

THE ANATOMY OF A BASIC C# CLASS

• Consider the following code:

```
using System; class
HelloClass
{
        public static int Main(string[] args)
        {
                Console.WriteLine("hello world\n"); return
                0;
        }
}
```

• All program-logic must be present within a type-definition.
• „Type" may be a member of the set {class, interface, structure, enumeration}.
• Every executable C# program must contain a *Main()* method within a class.
• *Main()* is used to signify entry point of the program.
• *Public* methods are accessible from other types.
• *Static* methods can be invoked directly from the class-level, without creating an object.
• Main() method has a single parameter, which is an array-of-strings (string[] args).
• args parameter may contain any number of incoming command-line arguments.
• *Console* class is defined within the *„System"* namespace.
• Console class contains method-named *WriteLine()*.
• *WriteLine()* is used to pump a text-string to the standard console.

PROCESSING COMMAND LINE PARAMETERS

Using Length property of System.Array

• Consider the following code:

```
using System; class
HelloClass
{
        public static int Main(string[] args)
        {
                Console.WriteLine(" Command line arguments \n "); for(int
                i=0;i<args.Length;i++)
                    Console.WriteLine("Argument:{0} ",args[i]);

                return 0;
        }
}
```

Output:

```
C:\> csc HelloClass.cs
C:\> HelloClass.exe three two one
Command line arguments
Argument: three Argument: two
Argument: one
```

• Here, we are checking to see if the array of strings contains some number of items using *Length* property of *System.Array*.

• If we have at least one member in the array, we loop over each item and print the contents to the output window.

Using "foreach" keyword

• Consider the following code:

```
using System; class
HelloClass
{
        public static int Main(string[] args)
        {
                Console.WriteLine(" Command line arguments \n ");
                foreach(string s in args) Console.WriteLine("Argument
                    {0}:",s);
        }
}
```

• *foreach* loop can used to iterate over all items within an array, without the need to test for the **array‟s upper limit.**

Using GetCommandLineArgs() method of System.Environment type

• Consider the following code:

```
using System; class
HelloClass
{
        public static int Main(string[] args)
        {
                string[] theArgs=Environment.GetCommandLineArgs();
                Console.WriteLine("path is :{0}",theArgs[0]); Console.WriteLine("
                Command line arguments \n ");
                for(int i=1;i<theArgs.Length;i++)
                        Console.WriteLine("Arguments :{0}",theArgs)
        }
}
```

Output:

C:\> csc HelloClass.cs

```
C:\> HelloClass.exe three two path is C:\>
HelloClass.cs Command line arguments
Argument: three
Argument: two
```

- We can also access command line arguments using the GetCommandLineArgs() method of the *System.Environment* type.
- First index identifies current directory containing the program itself, while remaining elements in the array contain the individual command-line arguments

UNDERSTANDING VALUE TYPES AND REFERENCE TYPES

- In .NET, data type may be value-based or reference-based.

VALUE TYPES	REFERENCE TYPES
• Value-based types include all numerical data types (int, float, char) as well as enumerations and structures. • These are allocated on the stack. • These can be quickly removed from memory once they fall out of the defining-scope. • By default, when we assign one value type to another, a member-by-member copy is achieved. • Consider the following code:	• Reference types include all strings & objects. • These are allocated on the garbage-collected heap. • These types stay in memory until the garbage collector destroys them. • By default, when we assign one reference type to another, a new reference to the same object in memory created. • Consider the following code:

struct Foo
{
public int x,y;
}
class ValRef
{
public static int Main(string[] args)
{
Foo f1=new Foo();
f1.x=100;
f1.y=100;
Console.WriteLine(“assigning f2 to f1”);
Foo f2=f1;
Console.WriteLine(“f1.x={0}”,f1.x);
Console.WriteLine(“f1.y={0}”,f1.y);
Console.WriteLine(“f2.y={0}”,f2.y);
Console.WriteLine(“f2.y={0}”,f2.y);
Console.WriteLine(“changing f2.x to 900”);
f2.x=900;
Console.WriteLine(“here are the X‟s again”);
Console.WriteLine(“f2.x={0}”,f2.x);
	class Foo
	{
	public int x,y;
	}
	class ValRef
	{
	public static int Main(string[] args)
	{
	Foo f1=new Foo();
	f1.x=100;
	f1.y=100;
	Console.WriteLine(“assigning f2 to f1”);
	Foo f2=f1;
	Console.WriteLine(“f1.x={0}”,f1.x);
	Console.WriteLine(“f1.y={0}”,f1.y);
	Console.WriteLine(“f2.y={0}”,f2.y);
	Console.WriteLine(“f2.y={0}”,f2.y);
	Console.WriteLine(“changing f2.x to 900”);
	f2.x=900;
	Console.WriteLine(“here are the X‟s again”);
	Console.WriteLine(“f2.x={0}”,f2.x);

Console.WriteLine("f1.x={0}",f1.x);
return 0;
}
}
	Console.WriteLine("f1.x={0}",f1.x);
	return 0;
	}
	}

Output: assigning f2 to f1 f1.x=100 f1.y=100 f2.x=100 f2.y=100 changing f2.x to 900 here are the X"sagain f2.x=900 f1.x=100	*Output:* assigning f2 to f1 f1.x=100 f1.y=100 f2.x=100 f2.y=100 changing f2.x to 900 here are the X"sagain f2.x=900 f1.x=900
• Here, we have 2 copies of the „Foo" type on the stack, each of which can be independently manipulated. Therefore, when we change the value of f2.x, the value of f1.x is unaffected.	• Here, we have 2 references to the same object in the memory. Therefore, when we change the value of f2.x, the value of f1.x is also changed.

VALUE TYPES	REFERENCE TYPES
Allocated on the stack	Allocated on the managed heap
Variables die when they fall out of the defining scope	Variables die when the managed heap is garbage collected
Variables are local copies	Variables are pointing to the memory occupied by the allocated instance
Variable are passed by value	Variables are passed by reference
Variables must directly derive from System.ValueType	Variables can derive from any other type as long as that type is not "sealed"
Value types are always sealed and cannot be extended	Reference type is not sealed, so it may function as a base to other types.
Value types are never placed onto the heap and therefore do not need to be finalized	Reference types finalized before garbage collection occurs

VALUE TYPES CONTAINING REFERENCE TYPES

- Consider the following code:

```
class TheRefType
{
        public string x;
        public TheRefType(string s)
        {x=s;}
}

struct InnerRef
{
        public TheRefType refType;
        public int structData; public
        InnerRef(string s)
        {
                refType=new TheRefType(s);
                structData=9;
        }
}

class ValRef
{
        public static int Main(string[] args)
        {
        Console.WriteLine("making InnerRef type and setting structData to 666"); InnerRef
        valWithRef=new InnerRef("initial value"); valWithRef.structData=666;

        Console.WriteLine("assigning valWithRef2 to valWithRef"); InnerRef
        valWithRef2;
        valWithRef2=valWithRef;

        Console.WriteLine("changing all values of valWithRef2");
        valWithRef2.refType.x="I AM NEW"; valWithRef2.structData=777;

        Console.WriteLine("values after change"); Console.WriteLine("valWithRef.refType.x is {0}",
        valWithRef.refType.x); Console.WriteLine("valWithRef2.refType.x is {0}",
        valWthRef2.refType.x); Console.WriteLine("valWithRef.structData is {0}",
        valWithRef.structData); Console.WriteLine("valWithRef2.structData is {0}",
        valWithRef2.structData);
        }
}
```

Output:

```
making InnerRef type and setting structData to 666
assigning valWithRef2 to valWithRef
changing all values of valWithRef2

values after change
valWithRef.refType.x is I AM NEW
valWithRef2.refType.x is I AM
NEW valWithRef.structData is 666
valWithRef2.structData is 777
```

- When a value-type contains other reference-types, assignment results in a copy of the references.

In this way, we have 2 independent structures, each of which contains a reference pointing to the same object in memory i.e. shallow copy**.**

- When we want to perform a **deep copy** (where the state of internal references is fully copied into a new object), we need to implement the ICloneable interface.

BOXING AND UN-BOXING

- We know that .NET defines 2 broad categories of data types viz. value-types and reference-types.
- Sometimes, we may need to convert variables of one category to the variables of other category. For doing so, .NET provides a mechanism called boxing.
- *Boxing* is the process of explicitly converting a value-type into a reference-type.
- When we *box* a variable, a new object is allocated in the heap and the value of variable is copied into the object.
- *Unboxing* is the process of converting the value held in the object reference back into a corresponding value type.
- When we try to unbox an object, the compiler first checks whether is the receiving data type is equivalent to the boxed type or not.
- If yes, the value stored in the object is copied into a variable in the stack.
- If we try to unbox an object to a data type other than the original type, an exception called InvalidCastException is generated.

- For example:

```
int p=20; object
ob=p;
--------
-------
int b=(int)ob; string          // unboxing successful
s=(string)ob;                  // InvalidCastException
```

- Generally, there will few situations in which we need boxing and/or unboxing.
- In most of the situations, C# compiler will automatically boxes the variables. For example, if we pass a value type data to a function having reference type object as a parameter, then automatic boxing takes place.

- Consider the following program:

```
using System;
class Test
{
        public static void MyFunc(object ob)
        {
            Console.WriteLine(ob.GetType());
            Console.WriteLine(ob.ToString());
            Console.WriteLine(((int)ob).GetTypeCode());            //explicit unboxing
        }

        public static void Main()
        {
                int x=20;
                MyFunc(x);          //automatic boxing
        }
}
```

<u>Output:</u>

```
System.Int32 20
Int32
```

- When we pass custom (user defined) structures/enumerations into a method taking generic System.Obejct parameter, we need to unbox the parameter to interact with the specific members of the structure/enumeration.

METHOD PARAMETER MODIFIERS

• Normally methods will take parameter. While calling a method, parameters can be passed in different ways.

• C# provides some parameter modifiers as shown:

Parameter Modifier	**Meaning**
(none)	If a parameter is not attached with any modifier, then parameter"s value is passed to the method. This is the default way of passing parameter. (call-by-value)
out	The output parameters are assigned by the called-method.
ref	The value is initially assigned by the caller, and may be optionally reassigned by the called-method
params	This can be used to send variable number of arguments as a single parameter. Any method can have only one *params* modifier and it should be the last parameter for the method.

THE DEFAULT PARAMETER PASSING BEHAVIOR

• By default, the parameters are passed to a method *by-value*.

• If we do not mark an argument with a parameter-centric modifier, a copy of the data is passed into the method.

• So, the changes made for parameters within a method will not affect the actual parameters of the calling method.

Consider the following program:

```
using System; class Test
{
        public static void swap(int x, int y)
        {
            int temp=x; x=y;
              y=temp;
        }

        public static void Main()
        {
                int x=5,y=20;
                Console.WriteLine("Before: x={0}, y={1}", x, y);
                swap(x,y);
                Console.WriteLine("After: x={0}, y={1}", x, y);
        }
}
```

Output:

```
Before: x=5, y=20 After : x=5, y=20
```

out KEYWORD

- Output parameters are assigned by the called-method.
- In some of the methods, we need to return a value to a calling-method. Instead of using *return*

statement, C# provides a modifier for a parameter as *out*.

- Consider the following program:

```
using System;
class Test
{
        public static void add(int x, int y, out int z)
        {
                z=x+y;
        }

        public static void Main()
        {
                int x=5,y=20, z;
                add(x, y, out z);
                Console.WriteLine("z={0}", z);
        }
 }
```

Output:
```
z=25
```

- Useful purpose of out: It allows the caller to obtain multiple return values from a single method-

invocation.

- Consider the following program:

```
using System;
class Test
{
        public static void MyFun(out int x, out string y, out bool z)
        {
                x=5;
                y="Hello, how are you?";
                 z=true;
        }

        public static void Main()
        {
                int a;
                string str;
                bool b;

                MyFun(out a, out str, out b);
                Console.WriteLine("integer={0} ", a);
                Console.WriteLine("string={0}", str);
                Console.WriteLine("boolean={0} ", b);

        }
}
```

Output:
```
integer=5,
string=Hello, how are you? boolean=true
```

ref KEYWORD

- The value is assigned by the caller but may be reassigned within the scope of the method-call.
- These are necessary when we wish to allow a method to operate on (and usually change the values of) various data points declared in the caller‟s scope.
- Differences between output and reference parameters:
  - **→ The *output* parameters do not need to be initialized before sending to called-method. Because it is assumed that the called-method will fill the value for such parameter.**
  - **→ The *reference* parameters must be initialized before sending to called-method. Because, we are passing a reference to an existing type and if we don‟t assign an initial value, it would be equivalent to working on NULL pointer.**
- Consider the following program:

```
using System;
class Test
{
        public static void MyFun(ref string s)
        {
                s=s.ToUpper();
        }

        public static void Main()
        {
                string s="hello";
                Console.WriteLine("Before:{0}",s);
                MyFun(ref s);
                Console.WriteLine("After:{0}",s);
        }

}
```

Output:
```
Before: hello
After: HELLO
```

params KEYWORD

- This can be used to send variable number of arguments as a single parameter.
- Any method can have only one *params* modifier and it should be the last parameter for the method
- Consider the following example:

```
using System;
class Test
{
        public static void MyFun(params int[] arr)
        {
                for(int i=0; i<arr.Length; i++)
                        Console.WriteLine(arr[i]);
        }

        public static void Main()
        {
                int[] a=new int[3]{5, 10, 15}; int
                p=25, q=102;

                MyFun(a);
                MyFun(p, q);
        }
}
```

```
Output
5    10   15    25    102
```

PASSING REFERENCE TYPES BY VALUE AND REFERENCE

- Consider the following program:

```
using System;
class Person
{
        string name; int
        age;
        public Person(string n, int a)
        {
                name=n;
                age=a;
        }

        public static void CallByVal(Person p)
        {
            p.age=66;
            p=new Person("John", 22);           //this will be forgotten after the call
        }


        public static void CallByRef(ref Person p)
        {
            p.age=66;
            p=new Person("John", 22);         // p is now pointing to a new object on the heap
        }

        public void disp()
        {
                Console.WriteLine("{0}        {1}", name, age);
        }

        public static void Main()
        {
                Person p1=new Person("Raja", 33);
                p1.disp();
                CallByVal(p1); p1.disp();
                CallByRef(ref p1);
                p1.disp();
        }


 }
```

Output:

```
Raja      33
Raja      66
John      22
```

- Rule1: If a reference type is passed by value, the called-method may change the values of the object‟s data but may not change the object it is referencing.
- Rule 2: If a reference type is passed by reference, the called-method may change the values of the object‟s data and also the object it is referencing.

ARRAY MANIPULATION IN C#

- C# arrays look like that of C/C++. But, basically, they are derived from the base class viz.

System.Array.

- *Array* is a collection of data elements of same type, which are accessed using numerical index.
- Normally, in C#, the array index starts with 0. But it is possible to have an array with arbitrary lower bound using the static method CreateInstance() of System.Array.
- Arrays can be single or multi-dimensional. The declaration of array would look like –

```
int[ ] a= new int[10];
a[0]= 5;
a[1]= 14;
……….
string[ ] s= new string[2]{"raja", "john"};
int[ ] b={15, 25, 31, 78}; //new is missing. Still valid
```

- In .NET, the members of array are automatically set to their respective default value. For example, in the statement,

    int[ ] a= new int[10];

all the elements of *a* are set to 0. Similarly, string array elements are set to *null* and so on.

ARRAY AS PARAMETERS AND RETURN VALUES

- Array can be passed as parameter to a method and also can be returned from a method.
- Consider the following program:

```
using System;
class Test
{
        public static void disp(int[ ] arr)            //taking array as parameter
        {
                for(int i=0;i<arr.Length;i++)
                    Console.WriteLine("{0} ", arr[i]);
        }

        public static string[ ] MyFun()                //returning an array
        {
                string[ ] str={"Hello", "World"}; return
                 str;
        }

        public static void Main()
        {
                int[ ] p=new int[ ]{20, 54, 12, -56}; disp(p);

                string[ ] strs=MyFun();
                foreach(string s in strs)
                        Console.WriteLine(s);
        }


}
```

Output-

```
20    54   12    -56  Hello      world
```

WORKING WITH MULTIDIMENSIONAL ARRAYS

- There are two types of multi-dimensional arrays in C# viz. rectangular array and jagged array.
- The rectangular array is an array of multiple dimensions and each row is of same length.
- Consider the following program:

```
using System;
class Test
{
        static void Main(string[]
        {
                // 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();
                }
        }
}
```

Output:

```
        0       0       0        0       0       0
        0       1       2        3       4       5
        0       2       4        6       8      10
        0       3       6        9      12      15
        0       4       8       12      16      20
        0       5      10       15      20      25
```

•A jagged array is an array whose elements are arrays.
The elements of array can be of different dimensions and sizes.
A jagged array is sometimes called an "array of arrays."

Unit-III
Object Oriented Programming

UNDERSTANDING OBJECT LIFETIME

• Automatic memory-management: Programmers never directly de-allocate an object from memory (therefore there is no "delete" keyword).

• Objects are allocated onto a region-of-memory termed the *'managed-heap'* where they will be automatically de-allocated by CLR at "sometime later".

• The golden rule of .NET Memory Management: "Allocate an object onto the managed-heap using the new keyword and forget about it".

• CLR removes the objects which are

→ **no longer needed or**

→ **unreachable by current application**

• Consider the following code:

```
//create a local car object
 public static int Main(string[] args)
 {
             //place an object onto the managed heap
             Car c=new Car("zen",200,100);
 }
 // if 'c' is the only reference to the
 //Car object, it may be destroyed when
 // Main exits
```

• Here, c is created within the scope of Main(). Thus, once the application shuts down, this reference is no longer valid and therefore it is a candidate for garbage collection.

• But, we cannot surely say that the object „c‟ is destroyed immediately after Main() function. All we can say is when CLR performs the next garbage collection; c is ready to be destroyed.

THE CIL OF "new"

• When C# compiler encounters the 'new' keyword, it will emit a CIL *"newobj"* instruction to the code- module.

• The *garbage-collector*(GC) is a tidy house-keeper.

GC compacts empty blocks of memory for purpose of optimization.

• The heap maintains a new-object-pointer(NOP).

NOP points to next available slot on the heap where the next object will be placed.

• The newobj instruction informs CLR to perform following sequence of events:

i) CLR calculates total memory required for the new object to be allocated (Figure 5.2).

If this object contains other internal objects, their memory is also taken into account.

Also, the memory required for each base class is also taken into account.

ii) Then, CLR examines heap to ensure that there is sufficient memory to store the new object.

If yes, the object's constructor is called and a reference to the object in the memory is returned (which just happens to be identical to the last position of NOP).

iii) Finally, CLR advances NOP to point to the next available slot on the heap.

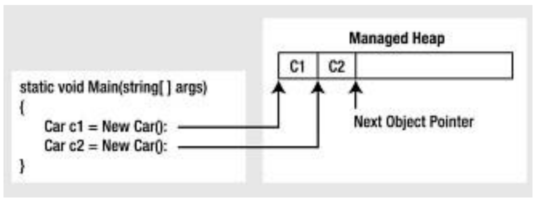


Figure 5.2: reference types are allocated on managed heap

DEFINING INTERFACES

• An interface is a collection of *abstract-methods* that may be implemented by a given class (in other words, an interface expresses a behavior that a given class/structure may choose to support).

• Interface is a pure protocol. i.e.

→ It never defines data-type &

→ It never provides a default implementation of the methods.

• Interface

→ never specifies a base class (not even System.Object) &

→ never contains member that do not take an access-modifier (as all interface-members are implicitly public).

• It can define any number of properties.

• Here is the formal definition:

```
public interface IPointy
{
        int GetNumberOfPoints();
}
```

• Interface-types are somewhat useless on their own (as they are nothing more than a named- **collection of abstract-members). Given this, we cannot allocate interface-types:**

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

IMPLEMENTING AN INTERFACE

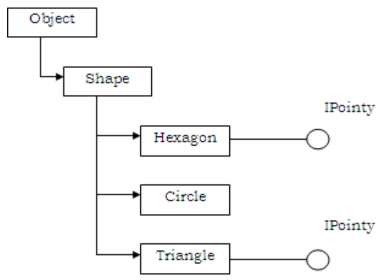
• A class/structure can extend its functionality by supporting a given interface using a comma- delimited list in the type-definition.

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

// This struct derives from System.ValueType and implements 2 interfaces.
public struct SomeStruct : ISomeInterface, IPointy
{      ...       }
```

• Implementing an interface is an *all-or-nothing* proposition i.e. the supporting-class cannot **selectively choose the members it wants to implement.**

• A given class may implement many interfaces **but the class may have exactly one base class.**



• Following figure illustrates IPointy-compatible classes.
• Consider the following code:

```
public class Hexagon: Shape, IPointy
{
        public Hexagon() {. . }
        public int GetNumberOfPoints()                    // IPointy Implementation.
        {        return 6;          }
}
public class Triangle: Shape, IPointy
{
        public Triangle() {. . }
        public int GetNumberOfPoints()                   // IPointy Implementation.
        {        return 3;          }
}
```

• Here, each class returns its number of points to the caller when asked to do so.

CONTRASTING INTERFACES TO ABSTRACT BASE CLASS

Abstract Base Classes

• Abstract base classes define a group of abstract-methods.
• They can
→ **define public, private & protected state data and**
→ **define any number of concrete-methods that can be accessed by the subclasses.**

Interfaces

• An interface is a collection of *abstract-methods* that may be implemented by a given class (in other words, an interface expresses a behavior that a given class/structure may choose to support).
• Interface is a pure protocol. i.e.
→ **It never defines data-type &**
→ **It never provides a default implementation of the methods.**
• Interface
→ **never specifies a base class (not even System.Object) &**
→ **never contains member that do not take an access-modifier (as all interface-members are implicitly public).**
• Consider the following code:

```
public interface IAmBadInterface
{
    // Error, interfaces can't define data!
    int myInt = 0;
    // Error, only abstract members allowed!
    void MyMethod()
    {
        Console.WriteLine("Hi!");
    }
}
```

• The interface-based protocol allows a given type to support numerous behaviors **while avoiding the issues that arise when deriving from multiple base classes.**
• Interface-based programming provides a way to add polymorphic behaviour into a system:

If multiple classes implement the same interface in their unique ways, we have the power to treat each type in the same manner.

INVOKING INTERFACE MEMBERS AT THE OBJECT LEVEL

• One way to interact with functionality-supplied by a given interface is to invoke the methods directly from the object-level.

```
public static void Main(string[] args)
{           // Call new Points member defined by IPointy
            Hexagon hex=new Hexagon();
            Console.WriteLine("points: {0}", hex.GetNumberOfPoints()); //prints 6
            Triangle tri=new Triangle();
            Console.WriteLine("points: {0}", tri.GetNumberOfPoints());           //prints 3
}
```

• Following 3 techniques can be used to check which interfaces are supported by a given type:

i. By explicit casting   ii. By using "as" keyword   iii. By using "is" keyword

Method 1: Explicit Casting

• An explicit-cast can be used to obtain an interface-reference.

• When we attempt to access an interface not supported by a given class using a direct-cast, the runtime throws an InvalidCastException.

```
        public static void Main(string[] args)
        {
                Hexagon hex=new Hexagon("bill"); IPointy
                ipt;
                try
                {
                        ipt=(IPointy)hex;                // explicit casting
                        Console.WriteLine(ipt.GetNumberOfPoints());
                }
                catch (InvalidCastException e)
                {
                        Console.WriteLine(e.Message);
                }
        }
```

Method 2: The "as" Keyword

• The "as" operator can be used to downcast between types or implemented-interface.

• The "as" keyword assigns the interface-variable to *null* if a given interface is not supported by the object (rather than throwing an exception).

```
        public static void Main(string[] args)
        {
                // Can we treat h as IPointy? Hexagon
                hex=new Hexagon("peter"); IPointy ipt = hex
                as IPointy;

                if(ipt != null)
                        Console.WriteLine(ipt.GetNumberOfPoints());
                else
                        Console.WriteLine("OOPS! Not pointy...");
        }
```

Method 3: The "is" Keyword

• The "is" operator can be used verify at runtime if an object is compatible with a given type.

```
        public static void Main(string[] args)
        {
               Triangle tri=new Triangle(); if(tri
               is IPointy)
                      Console.WriteLine(tri.GetNumberOfPoints());
               else
                       Console.WriteLine("OOPS! Not Pointy");
        }
```

• We may discover at runtime which items in the array support this behavior. For example,

```
public static void Main(string[] args)
{
      Shape[ ] s = { new Hexagon(), new Circle(), new Triangle("Ram"), new Circle("Sham")} ; for(int i = 0; i <
      s.Length; i++)
      {
                  // Recall the Shape base class defines an abstract Draw()
                  // member, so all shapes know how to draw themselves.
                 s[i].Draw();

                 if(s[i] is IPointy)
                       Console.WriteLine("Points: {0} ", ((IPointy)s[i]).GetNumberOfPoints());
                 else
                      Console.WriteLine("OOPS! Not Pointy");
      }
}
```

INTERFACES AS PARAMETERS

- Given that interfaces are valid .NET types, we may construct methods that take interfaces as parameters.
- To illustrate, assume we have defined another interface named IDraw3D:

```
// Models the ability to render a type in stunning 3D.
public interface IDraw3D
{
            void Draw3D();
}
```

- Next, assume that two of our three shapes (Circle and Hexagon) have been configured to support

<u>this new behavior:</u>

```
// Circle supports IDraw3D
public class Circle : Shape, IDraw3D
{
            public void Draw3D()
            {
                        Console.WriteLine("Drawing Circle in 3D!");
            }
}

// Hexagon supports IPointy and IDraw3D
public class Hexagon : Shape, IPointy, IDraw3D
{
            public void Draw3D()
            {
                        Console.WriteLine("Drawing Hexagon in 3D!");
            }
}
```

- If we now define a method taking an IDraw3D interface as a parameter, we are able to effectively send in *any* object implementing IDraw3D (if we attempt to pass in a type not supporting the necessary interface, we receive a compile-time error).

• Consider the following:

```
// Make some shapes. If they can be rendered in 3D, do it!
public class Program
{
        // I'll draw anyone supporting IDraw3D.
        public static void DrawIn3D(IDraw3D itf3d)
        {
                Console.WriteLine("Drawing IDraw3D compatible type");
                itf3d.Draw3D();
        }

        static void Main()
        {
                Shape[ ] s = { new Hexagon(), new Circle(), new Triangle()}; for(int i = 0; i <
                s.Length; i++)
                {
                        if(s[i] is IDraw3D)
                                DrawIn3D((IDraw3D)s[i]);
                }
        }
}
```

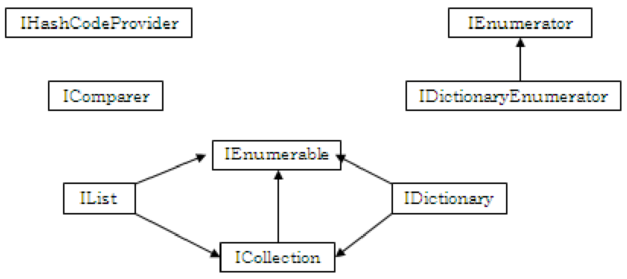
• Here, the triangle is never drawn, as it is not IDraw3D-compatible

EXPLORING THE SYSTEM.COLLECTIONS NAMESPACE

• System.Collections defines a numbers of standard interfaces. For example:

System.Collections Interface	**Meanin g**
ICollection	Defines generic characteristics (e.g., count and thread safety) for a collection type.
IComparer	Defines methods to support the comparison of objects for equality.
IDictionary	Allows an object to represent its contents using name/value pairs.
IDictionaryEnumerat or	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.
IList	Provides behavior to add, remove, and index items in a list of objects.

• Following figure illustrates the relationship between each type of the interfaces:



SYSTEM.COLLECTIONS.SPECIALIZED NAMESPACE

System.Collections.Specialized	Meanin g
CollectionsUtil	Creates collections that ignore the case in strings
HybridDictionary	Implements Dictionary by using a ListDictionary when the collection is small, and then switching to a Hashtable when the collection gets large
ListDictionary	Implements IDictionary using a singly linked-list. Recommended for collections that typically contain 10 items or less
NameObjectCollectionBase	Provides the abstract base class for a sorted collection of associated String keys and Object values that can be accessed either with the key or with the index
NameObjectCollectionB ase KeysCollection	Represent a collectio of the string keys of a s n collection
NameValueCollection	Represents a sorted collection of associated string- keys and string-values that can be accessed either with the key or with the index
StringCollection	Represents a collection of strings
StringDictionary	Implements a hash-table with the key strongly typed to be a string rather than an object
StringEnumerator	Supports a simple iteration over a StringCollection

ERRORS, BUGS & EXCEPTION

Keywords	Meaning
Errors	These are caused by end-user of the application. For e.g., entering un-expected value in a textbox, say USN.
Bugs	These are caused by the programmer. For e.g. → making use of NULL pointer or → referring array-index out of bound
Exceptions	These are regarded as runtime anomalies that are difficult to prevent. For e.g. → trying to connect non-existing database → trying to open a corrupted file → trying to divide a number by zero

THE ROLE OF .NET EXCEPTION HANDLING

• Programmers have a well-defined approach to error handling which is common to all .NET aware- languages.

• Identical syntax used to throw & catch exceptions across assemblies, applications & machine boundaries.

• Rather than receiving a cryptic-numerical value that identifies the problem at hand, exceptions are objects that contain a *human readable description* of the problem.

Objects also contain a detailed snapshot of the call-stack that eventually triggered the exception.

• The end-user can be provided with the *help-link information*.

Help-link information point to a URL that provides detailed information regarding error at hand.

THE ATOMS OF .NET EXCEPTION HANDLING

• Programming with structured exception handling involves the use of 4 key elements:

i) A 'type' that represents the details of the exception that occurred.

ii) A method that throws an instance of the exception-class to the caller.

iii) A block of code that will invoke the exception-ready method (i.e. try block).

iv) A block of code that will process the exception (i.e. catch block).

• C# offers 4 keywords {try, catch, throw and finally} that can be used to throw and handle exceptions.

• The type that represents the problem at hand is a class derived from *System.Exception*.

THE SYSTEM.EXCEPTION BASE CLASS

• All system-supplied and custom exceptions derive from the System.Exception base class (which in turn derives from System.Object).

```
public class Exception: object
{
        public Exception();
        public Exception(string message);
        public string HelpLink { virtual get; virtual set;} public
        string Message {virtual get;}
        public string Source {virtual get; virtual set;} public
        MethodBase TargetSite{get;}
}
```

CORE MEMBERS OF SYSTEM.EXCEPTION TYPE

Member	Meaning
Message	This returns textual description of a given error. The error-message itself is set as a constructor-parameter.
TargetSite	This returns name of the method that threw the exception.
Source	This returns name of the assembly that threw the exception.
HelpLink	This returns a URL to a help-file which describes the error in detail.
StackTree	This returns a string that identifies the sequence of calls that triggered the exception.
InnerException	This is used to obtain information about the previous exceptions that causes the current exception to occur.

THROWING A GENERIC EXCEPTION

• During the program, if any exception occurs, we can throw a specific exception like

→ **DivideByZeroException**

→ **FileNotFoundException**

→ **OutOfMemoryException**

• The object of Exception class can handle any type of exception, as it is a base class for all type of exceptions.

• Consider the following code:

```
using System;
class Test
{
        int Max=100;

        public void Fun(int d)
        {
                if(d>Max)
                   throw new Exception("crossed limit!!!"); else
                  Console.WriteLine("speed is ={0}", d);
        }

        public static void Main()
        {
                        Test ob=new Test(); Console.WriteLine("Enter a
                        number:");
                                int d=int.Parse(Console.ReadLine());

                        ob.Fun(d);
         }
}
```

Output:

```
Enter a number: 12 speed is
=12
Enter a number: 567
Unhandled Exception: System.Exception: crossed limit!!! at
Test.Fun(Int32 d)
at Test.Main()
```

- In the above example, if the entered-value d is greater than 100, then we throw an exception.
- Firstly, we have created a new instance of the System.Exception class,
  then we have passed a message "crossed limit" to a Message property of Exception class.
- It is upto the programmer to decide exactly
  → **what constitute an exception &**
  → **when the exception should be thrown.**
- Using *throw* keyword, program throws an exception when a problem shows up.

CATCHING EXCEPTIONS

- A *catch* block contains a code that will process the exception.
- When the program throws an exception, it can be caught using "catch" keyword.
- Once the exception is caught, the members of System.Exception class can be invoked.
- These members can be used to
  → **display a message about the exception**
  → **store the information about the error in a file and**
  → **send a mail to administrator**
- Consider the following code:

```
using System;
class Test
{
        int Max=100;

        public void Fun(int d)
        {
                    try
                    {
                        if(d>Max)
                            throw new Exception("crossed limit!!!");
                    }
                    catch(Exception e)
                    {
                            Console.WriteLine("Message:{0}",e.Message);
                            Console.WriteLine("Method:{0}",e.TargetSite);
                    }

                        //the error has been handled,
                        //continue with the flow of this application
                    Console.WriteLine("Speed is ={0}", d);
        }

        public static void Main()
        {
                Test ob=new Test();

                Console.WriteLine("Enter a number:"); int
                d=int.Parse(Console.ReadLine());

                ob.Fun(d);
        }
}
```

```
Output-1:
        Enter a number: 12
        Speed is =12

Output-2:
        Enter a number: 123 Message:
        crossed limit!!! Method: Void
        Fun(Int32) Speed is=123
```

- A *try* block contains a code that will check for any exception that may be encountered during its scope.

• If an exception is detected, the program control is sent to the appropriate catch-block. Otherwise, the catch-block is skipped.

• Once an exception is handled, the application will continue its execution from very next point after catch-block.

THE FINALLY BLOCK

• The try/catch block may also be attached with an optional "finally" block.

• A *finally* block contains a code that is executed "*always*", irrespective of whether an exception is thrown or not.

• For example, if you open a file, it must be closed whether an exception is raised or not.

• The finally block is used to

→ **clean-up any allocated memory**
→ **close a database connection**
→ **close a file which is in use**

• The code contained within a 'finally' block executes "all the time", even if the logic within try clause does not generate an exception.

• Consider the following code:

```
using System; class
DivNumbers
{
        public static int SafeDivision(int num1, int num2)
        {
                if(num2==0)
                      throw new DivideByZeroException("you cannot divide a number by 0");

                return num1/num2;
        }

        static void Main(string[] args)
        {
            int num1=15, num2=0; int
            result=0;

            try
            {
                      result=SafeDivision(num1,num2);
            }
            catch( DivideByZeroException e)
            {
                      Console.WriteLine("Error message = {0}", e.Message);
            }
            finally
            {
                      Console.WriteLine("Result: {0}", result);
            }
        }
}
```

Output:

```
        Error message = you cannot divide a number by 0 Result: 0
```

UNIT-IV
Delegates And Events

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.

To understand the process, here is the crux of the generated BinaryOp class type (**bold** marks the items specified by the defined delegate type):

```
sealed class BinaryOp : System.MulticastDelegate
{
public BinaryOp(object target, uint functionAddress);
public void Invoke(int x, int y);
public IAsyncResult BeginInvoke(int x, int y,
AsyncCallback cb, object state);
public int EndInvoke(IAsyncResult result);
}
```

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.

The Role of .NET Assemblies

.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 Are Configurable

Assemblies can be deployed as “private” or “shared.” Private assemblies reside in the same directory (or possibly a subdirectory) as the client application making use of them. Shared assemblies, on the other hand, are libraries intended to be consumed by numerous applications on a single machine and are deployed to a specific directory termed the *Global Assembly Cache* (*GAC*). Regardless of how you deploy your assemblies, you are free to author XML-based configuration files. Using these configuration files, the CLR can be instructed to “probe” for assemblies under a specific location, load a specific version of a referenced assembly for a particular client, or consult an arbitrary directory on your local machine, your network location, or a web-based URL.

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

Single-File and Multifile Assemblies
Technically speaking, an assembly can be composed of multiple *modules*. Amodule is really nothing more than a generic term for a valid .NET binary file. In most situations, an assembly is in fact composed of a single module. In this case, there is a one-to-one correspondence between the (logical) assembly and the underlying (physical) binary (hence the term *single-file assembly*).

Single-file assemblies contain all of the necessary elements (header information, CIL code, type metadata, manifest, and required resources) in a single *.exe or *.dll package. Figure 11-3 illustrates the composition of a single-file assembly.

A multifile assembly, on the other hand, is a set of .NET *.dlls that are deployed and versioned as a single logic unit. Formally speaking, one of these *.dlls is termed the primary module and contains the assembly-level manifest (as well as any necessary CIL code, metadata, header information, and optional resources). The manifest of the primary module records each of the related *.dll files it is dependent upon.

Building aVisual 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.

Like C#, Visual Basic .NET requires you to list each namespace used within the current file. However, Visual Basic .NET offers the Imports keyword rather than the C# using keyword. Given this, add the following Imports statement within the Module1.vb code file:

```
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()
{
}
}
```

Building and Consuming a Multifile Assembly

Now that you have constructed and consumed a single-file assembly, let's examine the process of building a multifile assembly. Recall that amultifile assembly is simply a collection of related modules that is deployed and versioned as a single unit. At the time of this writing, Visual Studio 2005 does not support a C# multifile assembly project template. Therefore, you will need to make use of the command-line compiler (csc.exe) if you wish to build such as beast.

To illustrate the process, you will build a multifile assembly named AirVehicles. The primary module (airvehicles.dll) will contain a single class type named Helicopter. The related manifest (also contained in airvehicles.dll) catalogues an additional *.netmodule file named ufo.netmodule, which contains another class type named (of course) Ufo. Although both class types are physically contained in separate binaries, you will group them into a single namespace named AirVehicles.

Finally, both classes are created using C# (although you could certainly mix and match languages if you desire).

To begin, open a simple text editor (such as Notepad) and create the following Ufo class definition saved to a file named ufo.cs:

```
using System;
namespace AirVehicles
{
public class Ufo
{
public void AbductHuman()
{
Console.WriteLine("Resistance is futile");
}
}
}
```

To compile this class into a .NET module, navigate to the folder containing ufo.cs and issue the following command to the C# compiler (the module option of the /target flag instructs csc.exe to produce a *.netmodule as opposed to a *.dll or *.exe file):

```
csc.exe /t:module ufo.cs
```

Understanding Private Assemblies

Technically speaking, the assemblies you've created thus far in this chapter have been deployed as *private assemblies*. Private assemblies are required to be located within the same directory as the client application (termed the *application directory*) or a subdirectory thereof. Recall that when you set a reference to CarLibrary.dll while building the CSharpCarClient.exe and VbNetCarClient.exe applications, Visual Studio 2005 responded by placing a copy of CarLibrary.dll within the client's application directory.

When a client program uses the types defined within this external assembly, the CLR simply loads the local copy of CarLibrary.dll. Because the .NET runtime does not consult the system registry when searching for referenced assemblies, you can relocate the CSharpCarClient.exe (or VbNetCarClient.exe) and CarLibrary.dll assemblies to location on your machine and run the application (this is often termed *Xcopy deployment*).

Uninstalling (or replicating) an application that makes exclusive use of private assemblies is a no-brainer: simply delete (or copy) the application folder. Unlike with COM applications, you do not need to worry about dozens of orphaned registry settings. More important, you do not need to worry that the removal of private assemblies will break any other applications on the machine.

Understanding Publisher Policy Assemblies
The next configuration issue you'll examine is the role of *publisher policy assemblies*. As you've just seen, *.config files can be constructed to bind to a specific version of a shared assembly, thereby bypassing the version recorded in the client manifest. While this is all well and good, imagine you're an administrator who now needs to reconfigure all client applications on a given machine to rebind to version 2.0.0.0 of the CarLibrary.dll assembly. Given the strict naming convention of a configuration file, you would need to duplicate the same XML content in numerous locations (assuming you are, in fact, aware of the locations of the executables using CarLibrary!). Clearly this would be a maintenance nightmare.

Publisher policy allows the publisher of a given assembly (you, your department, your company, or what have you) to ship a binary version of a *.config file that is installed into the GAC along with the newest version of the associated assembly. The benefit of this approach is that client application directories do *not* need to contain specific *.config files. Rather, the CLR will read the current manifest and attempt to find the requested version in the GAC. However, if the CLR finds a publisher policy assembly, it will read the embedded XML data and perform the requested redirection *at the level of the GAC.*

Publisher policy assemblies are created at the command line using a .NET utility named al.exe (the assembly linker). While this tool provides a large number of options, building a publisher policy assembly requires you only to pass in the following input parameters:
• The location of the *.config or *.xml file containing the redirecting instructions
• The name of the resulting publisher policy assembly
• The location of the *.snk file used to sign the publisher policy assembly
• The version numbers to assign the publisher policy assembly being constructed
If you wish to build a publisher policy assembly that controls CarLibrary.dll, the command set is as follows:

```
al /link: CarLibraryPolicy.xml /out:policy.1.0.CarLibrary.dll
/keyf:C:\MyKey\myKey.snk /v:1.0.0.0
```

Understanding the <codeBase> Element
Application configuration files can also specify *code bases*. The <codeBase> element can be used to instruct the CLR to probe for dependent assemblies located at arbitrary locations (such as network share points, or simply a local directory outside a client's application directory).

The System.Configuration Namespace
Currently, all of the *.config files shown in this chapter have made use of well-known XML elements that are read by the CLR to resolve the location of external assemblies. In addition to these recognized elements, it is perfectly permissible for a client configuration file to

contain application-specific data that has nothing to do with binding heuristics. Given this, it should come as no surprise that the .NET Framework provides a namespace that allows you to programmatically read the data within a client configuration file.
The System.Configuration namespace provides a small set of types you may use to read custom data from a client's *.config file. These custom settings must be contained within the scope of an <appSettings> element. The <appSettings> element contains any number of <add> elements that define a key/value pair to be obtained programmatically.

For example, assume you have a *.config file for a console application named AppConfigReaderApp
that defines a database connection string and a point of data named timesToSayHello:

```
<configuration>
<appSettings>
<add key="appConStr"
value="server=localhost;uid='sa';pwd='';database=Cars" />
<add key="timesToSayHello" value="8" />
</appSettings>
</configuration>
```

Reading these values for use by the client application is as simple as calling the instance-level GetValue() method of the System.Configuration.AppSettingsReader type. As shown in the following code, the first parameter to GetValue() is the name of the key in the *.config file, whereas the second parameter is the underlying type of the key (obtained via the C# typeof operator):

```
class Program
{
static void Main(string[] args)
{
// Create a reader and get the connection string value.
AppSettingsReader ar = new AppSettingsReader();
Console.WriteLine(ar.GetValue("appConStr", typeof(string)));
// Now get the number of times to say hello, and then do it!
int numbOfTimes = (int)ar.GetValue("timesToSayHello", typeof(int));
for(int i = 0; i < numbOfTimes; i++)
Console.WriteLine("Yo!");
Console.ReadLine();
}
}
```

The AppSettingsReader class type does *not* provide a way to write application-specific data to a *.config file. While this may seem like a limitation at first encounter, it actually makes good sense. The whole idea of a *.config file is that it contains read-only data that is consulted by the CLR (or possibly the AppSettingsReader type) after an application has already been deployed to a target machine.

UNIT-V
ADO.NET

A High-Level Definition of ADO.NET
If you have a background in Microsoft's previous COM-based data access model (Active Data Objects,or ADO), understand that ADO.NET has very little to do with ADO beyond the letters "A," "D," and "O." While it is true that there is some relationship between the two systems (e.g., each has the concept of connection and command objects), some familiar ADO types (e.g., the Recordset) no longer exist. Furthermore, there are a number of new ADO.NET types that have no direct equivalent under classic ADO (e.g., the data adapter).
Unlike classic ADO, which was primarily designed for tightly coupled client/server systems.

ADO.NET was built with the disconnected world in mind, using DataSets. This type represents a local copy of any number of related tables. Using the DataSet, the client tier is able to manipulate and update its contents while disconnected from the data source, and it can submit the modified data back for processing using a related data adapter.

Another major difference between classic ADO and ADO.NET is that ADO.NET has deep support for XML data representation. In fact, the data obtained from a data store is serialized (by default) as XML. Given that XML is often transported between layers using standard HTTP, ADO.NET is not limited by firewall constraints.

The Two Faces of ADO.NET
The ADO.NET libraries can be used in two conceptually unique manners: connected or disconnected. When you are making use of the *connected layer*, your code base will explicitly connect to and disconnect from the underlying data store. When you are using ADO.NET in this manner, you typically interact with the data store using connection objects, command objects, and data reader objects. As you will see later in this chapter, data readers provide a way to pull records from a data store using a forward-only, read-only approach (much like a fire-hose cursor).
The disconnected layer, on the other hand, allows you to obtain a set of DataTable objects (contained within a DataSet) that functions as a client-side copy of the external data. When you obtain a DataSet using a related data adapter object, the connection is automatically opened and closed on your behalf. As you would guess, this approach helps quickly free up connections for other callers. Once the client receives a DataSet, it is able to traverse and manipulate the contents without incurring the cost of network traffic. As well, if the client wishes to submit the changes back to the data store, the data adapter (in conjunction with a set of SQL statements) is used once again to update the data source, at which point the connection is closed immediately.

Understanding ADO.NET Data Providers
ADO.NET does not provide a single set of types that communicate with multiple database management systems (DBMSs). Rather, ADO.NET supports multiple *data providers*, each of which is optimized to interact with a specific DBMS. The first benefit of this approach is that a specific data provider can be programmed to access any unique features of the DBMS. Another benefit is that a specific data provider is able to directly connect to the underlying engine of the DBMS without an intermediate mapping layer standing between the tiers.

Simply put, a *data provider* is a set of types defined in a given namespace that understand how to communicate with a specific data source. Regardless of which data provider you make use of, each defines a set of class types that provide core functionality. Table 22-1 documents some (but not all) of the core common objects, their base class (all defined in the System.Data.Common namespace), and their implemented data-centric interfaces (each defined in the System.Data namespace).

Object	**Base Class**	**Implemented Interfaces**	**Meaning in Life**
Connection	**DbConnection**	**IDbConnection**	Provides the ability to connect to anddisconnect from the data store. Connection objects also provide access to a related transaction object.
Command	**DbCommand**	**IDbCommand**	Represents a SQL query or name of a stored procedure. Command objects also provide access to the provider's data reader object.
DataReader	**DbDataReader**	**IDataReader, IDataRecord**	Provides forward-only, read-only access to data.
DataAdapter	**DbDataAdapter**	**IDataAdapter, IDbDataAdapter**	Transfers DataSets between the caller and the data store. Data adapters contain a set of four internal command objects used to select,insert, update, and delete information from the data store.
Parameter	**DbParameter**	**IDataParameter**, IDbDataParameter	Represents named parameter within a parameterized query.
Transaction	**DbTransaction**	**IDbTransaction**	Performs a database transaction.

Additional ADO.NET Namespaces

In addition to the .NET namespaces that define the types of a specific data provider, the base class

Namespace	Meaning in Life
Microsoft.SqlServer.Server	This new .NET 2.0 namespace provides types that allow you to author stored procedures via managed languages for SQL Server 2005.
System.Data	This namespace defines the core ADO.NET types used by all data providers.
System.Data.Common	This namespace contains types shared between data providers, including the .NET 2.0 data provider factory types.
System.Data.Design	This new .NET 2.0 namespace contains various types used to construct a design-time appearance for custom data components.
System.Data.Sql	This new .NET 2.0 namespace contains types that allow you to discover Microsoft SQL Server instances installed on the current local network.
System.Data.SqlTypes	This namespace contains native data types used by Microsoft SQL Server. Although you are always free to use the corresponding CLR data types, the SqlTypes are optimized to work with SQL Server.

The System.Data Types

Of all the ADO.NET namespaces, System.Data is the lowest common denominator. You simply cannot build ADO.NET applications without specifying this namespace in your data access applications.

This namespace contains types that are shared among all ADO.NET data providers, regardless of the underlying data store. In addition to a number of database-centric exceptions (NoNullAllowedException, RowNotInTableException, MissingPrimaryKeyException, and the like), System.Data contains types that represent various database primitives (tables, rows, columns, constraints, etc.), as well as the common interfaces implemented by data provider objects.

Type	**Meaning in Life**
Constraint	Represents a constraint for a given DataColumn object
DataColumn	Represents a single column within a DataTable object
DataRelation	Represents a parent/child relationship between two DataTable objects
DataRow	Represents a single row within a DataTable object
DataSet	Represents an in-memory cache of data consisting of any number of interrelated DataTable objects
DataTable	Represents a tabular block of in-memory data
DataTableReader	Allows you to treat a DataTable as a fire-hose cursor (forward only, read-only data access); new in .NET 2.0
DataView	Represents a customized view of a DataTable for sorting, filtering, searching, editing, and navigation
IDataAdapter	Defines the core behavior of a data adapter object

Abstracting Data Providers Using Interfaces

At this point, you should have a better idea of the common functionality found among all .NET data providers. Recall that even though the exact names of the implementing types will differ among data providers, you are able to program against these types in a similar manner—that's the beauty of interface-based polymorphism. Therefore, if you define a method that takes an IDbConnection parameter, you can pass in any ADO.NET connection object:

```
public static void OpenConnection(IDbConnection cn)
{
// Open the incoming connection for the caller.
cn.Open();
}
```

The same holds true for a member return value. For example, consider the following simple C# program, which allows the caller to obtain a specific connection object using the value of a custom enumeration (assume you have "used" System.Data):

```
namespace ConnectionApp
{
enum DataProvider
{ SqlServer, OleDb, Odbc, Oracle }
class Program
{
static void Main(string[] args)
{
// Get a specific connection.
IDbConnection myCn = GetConnection(DataProvider.SqlServer);
// Assume we wish to connect to the SQL Server Pubs database.
myCn.ConnectionString =
"Data Source=localhost;uid=sa;pwd=;Initial Catalog=Pubs";

// Now open connection via our helper function.
OpenConnection(myCn);
// Use connection and close when finished.
...
myCn.Close();
}
static IDbConnection GetConnection(DataProvider dp)
{
IDbConnection conn = null;
switch (dp)
{
case DataProvider.SqlServer:
conn = new SqlConnection();
break;
case DataProvider.OleDb:
conn = new OleDbConnection();
break;
case DataProvider.Odbc:
conn = new OdbcConnection();
break;
case DataProvider.Oracle:
conn = new OracleConnection();
break;
}
return conn;
}
}
}
```

The benefit of working with the general interfaces of System.Data is that you have a much better chance of building a flexible code base that can evolve over time.

The .NET 2.0 Provider Factory Model

Under NET 2.0, we are now offered a data provider factory pattern that allows us to build a single code base using generalized data access types. Furthermore, using application configuration files (and the spiffy new <connectionStrings> section), we are able to obtain providers and connection strings declaratively without the need to recompile or redeploy the client software.

To understand the data provider factory implementation, recall from Table 22-1 that the objects within a data provider each derive from the same base classes defined within the System.Data.Common namespace:

• DbCommand: Abstract base class for all command objects
• DbConnection: Abstract base class for all connection objects
• DbDataAdapter: Abstract base class for all data adapter objects
• DbDataReader: Abstract base class for all data reader objects
• DbParameter: Abstract base class for all parameter objects
• DbTransaction: Abstract base class for all transaction objects

In addition, as of .NET 2.0, each of the Microsoft-supplied data providers now provides a specific class deriving from System.Data.Common.DbProviderFactory. This base class defines a number of methods that retrieve provider-specific data objects. Here is a snapshot of the relevant members of DbProviderFactory:

```
public abstract class DbProviderFactory
{
...
public virtual DbCommand CreateCommand();
public virtual DbCommandBuilder CreateCommandBuilder();
public virtual DbConnection CreateConnection();
public virtual DbConnectionStringBuilder CreateConnectionStringBuilder();
public virtual DbDataAdapter CreateDataAdapter();
public virtual DbDataSourceEnumerator CreateDataSourceEnumerator();
public virtual DbParameter CreateParameter();
}
```

To obtain the DbProviderFactory-derived type for your data provider, the System.Data.Common namespace provides a class type named DbProviderFactories (note the plural in this type’s name). Using the static GetFactory() method, you are able to obtain the specific (which is to say, singular) DbProviderFactory of the specified data provider,
for example:

```
static void Main(string[] args)
{
// Get the factory for the SQL data provider.
DbProviderFactory sqlFactory =
DbProviderFactories.GetFactory("System.Data.SqlClient");
...
// Get the factory for the Oracle data provider.
DbProviderFactory oracleFactory =
DbProviderFactories.GetFactory("System.Data.OracleClient");
...
}
```

As you might be thinking, rather than obtaining a factory using a hard-coded string literal, you could read in this information from a client-side *.config
However, in any case, once you have obtained the factory for your data provider, you are able to obtain the associated provider-specific
data objects (connections, commands, etc.).

Understanding the Connected Layer of ADO.NET

Recall that the *connected layer* of ADO.NET allows you to interact with a database using the connection,
command, and data reader objects of your data provider. Although you have already made use
of these objects in the previous DataProviderFactory example, let's walk through the process once
again in detail. When you wish to connect to a database and read the records using a data reader
object, you need to perform the following steps:

1. Allocate, configure, and open your connection object.
2. Allocate and configure a command object, specifying the connection object as a constructor
argument or via the Connection property.
3. Call ExecuteReader() on the configured command object.
4. Process each record using the Read() method of the data reader.

To get the ball rolling, create a brand-new console application named CarsDataReader. The goal is to open a connection (via the SqlConnection object) and submit a SQL query (via the SqlCommand object) to obtain all records within the Inventory table of the Cars database. At this point, you will use a SqlDataReader to print out the results using the type indexer. Here is the complete code within Main(), with analysis to follow:

```
class Program
{
static void Main(string[] args)
{
Console.WriteLine("***** Fun with Data Readers *****\n");
// Create an open a connection.
SqlConnection cn = new SqlConnection();
cn.ConnectionString =
"uid=sa;pwd=;Initial Catalog=Cars; Data Source=(local)";
cn.Open();
// Create a SQL command object.
string strSQL = "Select * From Inventory";
SqlCommand myCommand = new SqlCommand(strSQL, cn);
// Obtain a data reader a la ExecuteReader().
SqlDataReader myDataReader;
myDataReader = myCommand.ExecuteReader(CommandBehavior.CloseConnection);
// Loop over the results.
while (myDataReader.Read())
{
Console.WriteLine("-> Make: {0}, PetName: {1}, Color: {2}.",
myDataReader["Make"].ToString().Trim(),
myDataReader["PetName"].ToString().Trim(),
myDataReader["Color"].ToString().Trim());
}
// Because we specified CommandBehavior.CloseConnection, we
// don't need to explicitly call Close() on the connection.
myDataReader.Close();
}
}
```

Working with Data Readers

Once you have established the active connection and SQL command, the next step is to submit the query to the data source. As you might guess, you have a number of ways to do so. The DbDataReader type (which implements IDataReader) is the simplest and fastest way to obtain information from a data store. Recall that data readers represent a read-only, forward-only stream of data returned one record at a time. Given this, it should stand to reason that data readers are useful only when submitting SQL selection statements to the underlying data store.

Data readers are useful when you need to iterate over large amounts of data very quickly and have no need to maintain an in-memory representation. For example, if you request 20,000 records from a table to store in a text file, it would be rather memory-intensive to hold this information in a DataSet. A better approach is to create a data reader that spins over each record as rapidly as possible.

Be aware, however, that data reader objects (unlike data adapter objects, which you'll examine later) maintain an open connection to their data source until you explicitly close the session.

Data reader objects are obtained from the command object via a call to ExecuteReader(). When invoking this method, you may optionally instruct the reader to automatically close down the related connection object by specifying CommandBehavior.CloseConnection.

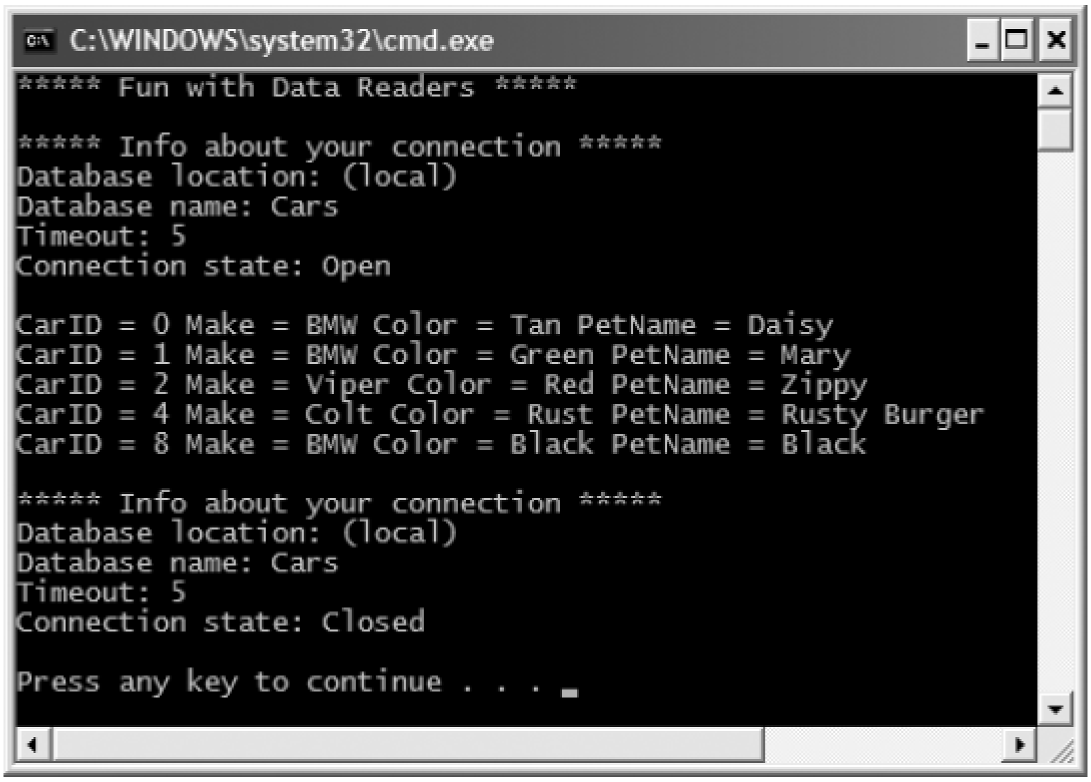
The following use of the data reader leverages the Read() method to determine when you have reached the end of your records (via a false return value). For each incoming record, you are making use of the type indexer to print out the make, pet name, and color of each automobile. Also note that you call Close() as soon as you are finished processing the records, to free up the connection object:

```
static void Main(string[] args)
{
...
// Obtain a data reader a la ExecuteReader().
SqlDataReader myDataReader;
myDataReader = myCommand.ExecuteReader(CommandBehavior.CloseConnection);
// Loop over the results.
while (myDataReader.Read())
{
Console.WriteLine("-> Make: {0}, PetName: {1}, Color: {2}.",
myDataReader["Make"].ToString().Trim(),
myDataReader["PetName"].ToString().Trim(),
myDataReader["Color"].ToString().Trim());
}
myDataReader.Close();
ShowConnectionStatus(cn);
}
```

The indexer of a data reader object has been overloaded to take either a string (representing the name of the column) or an integer (representing the column's ordinal position). Thus, you could clean up the current reader logic (and avoid hard-coded string names) with the following update (note the use of the FieldCount property):

```
while (myDataReader.Read())
{
Console.WriteLine("***** Record *****");
for (int i = 0; i < myDataReader.FieldCount; i++)
{
Console.WriteLine("{0} = {1} ",
myDataReader.GetName(i),
myDataReader.GetValue(i).ToString().Trim());
}
Console.WriteLine();
}
```

If you compile and run your project, you should be presented with a list of all automobiles in the Inventory table of the Cars database



```
***** Fun with Data Readers *****

***** Info about your connection *****
Database location: (local)
Database name: Cars
Timeout: 5
Connection state: Open

CarID = 0 Make = BMW Color = Tan PetName = Daisy
CarID = 1 Make = BMW Color = Green PetName = Mary
CarID = 2 Make = Viper Color = Red PetName = Zippy
CarID = 4 Make = Colt Color = Rust PetName = Rusty Burger
CarID = 8 Make = BMW Color = Black PetName = Black

***** Info about your connection *****
Database location: (local)
Database name: Cars
Timeout: 5
Connection state: Closed

Press any key to continue . . .
```

Understanding the Disconnected Layer of ADO.NET

As you have seen, working with the connected layer allows you to interact with a database using connection, command, and data reader objects. With this small handful of types, you are able to select, insert, update, and delete records to your heart’s content (as well as trigger stored procedures).

In reality, however, you have seen only half of the ADO.NET story. Recall that the ADO.NET object model can be used in a *disconnected* manner.When you work with the disconnected layer of ADO.NET, you will still make use of connection and command objects. In addition, you will leverage a specific object named a *data adapter* (which extends the abstract DbDataAdapter) to fetch and update data. Unlike the connected layer, data obtained via a data adapter is not processed using data reader objects. Rather, data adapter objects make use of DataSet objects to move data between the caller and data source. The DataSet type is a container for any number of DataTable objects, each of which contains a collection of DataRow and DataColumn objects.

The data adapter object of your data provider handles the database connection automatically. In an attempt to increase scalability, data adapters keep the connection open for the shortest possible amount of time. Once the caller receives the DataSet object, he is completely disconnected from the DBMS and left with a local copy of the remote data. The caller is free to insert, delete, or update rows from a given DataTable, but the physical database is not updated until the caller explicitly passes the DataSet to the data adapter for updating. In a nutshell, DataSets allow the clients to pretend they are indeed always connected, when in fact they are operating on an in-memory database

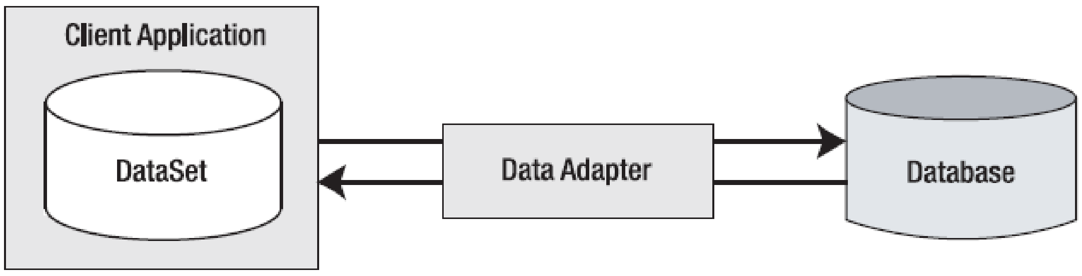


Figure 22-10. *Data adapter objects move* DataSets *to and from the client tier.*

Given that the centerpiece of the disconnected layer is the DataSet type, your next task is to learn how to manipulate a DataSet manually. Once you understand how to do so, you will have no problem manipulating the contents of a DataSet retrieved from a data adapter object.

Understanding the Role of the DataSet

Simply put, a DataSet is an in-memory representation of external data. More specifically, a DataSet is a class type that maintains three internal strongly typed collections

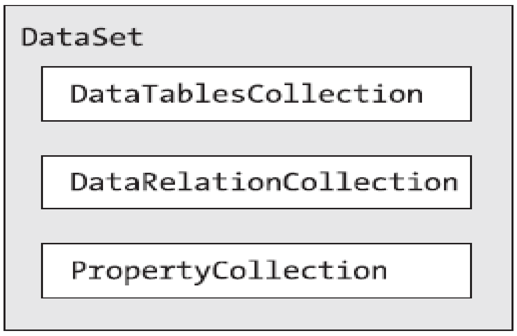


Figure 22-11. *The anatomy of a* DataSet

The Tables property of the DataSet allows you to access the DataTableCollection that contains the individual DataTables. Another important collection used by the DataSet is the DataRelationCollection.Given that a DataSet is a disconnected version of a database schema, it can programmatically represent the parent/child relationships between its tables. For example, a relation can be created between two tables to model a foreign key constraint using the DataRelation type. This object can then be added to the DataRelationCollection through the Relations property. The ExtendedProperties property provides access to the PropertyCollection object, which allows you to associate any extra information to the DataSet as name/value pairs. This information can literally be anything at all, even if it has no bearing on the data itself. For example, you can associate your company's name to a DataSet, which can then function as in-memory metadata. Other examples of extended properties might include timestamps, an encrypted password that must be supplied to access the contents of the DataSet, a number representing a data refresh rate, and so forth.

Table 22-9. *Properties of the Mighty* DataSet

Property	Meaning in Life
CaseSensitive	Indicates whether string comparisons in DataTable objects are case sensitive (or not).
DataSetName	Represents the friendly name of this DataSet. Typically this value is established as a constructor parameter.
EnforceConstraints	Gets or sets a value indicating whether constraint rules are followed when attempting any update operation.
HasErrors	Gets a value indicating whether there are errors in any of the rows in any of the DataTables of the DataSet.
RemotingFormat	This new .NET 2.0 property allows you to define how the DataSet should serialize its content (binary or XML) for the .NET remoting layer.

Table 22-10. *Methods of the Mighty* DataSet

Methods	Meaning in Life
AcceptChanges()	Commits all the changes made to this DataSet since it was loaded or the last time AcceptChanges() was called.
Clear()	Completely clears the DataSet data by removing every row in each DataTable.
Clone()	Clones the structure of the DataSet, including all DataTables, as well as all relations and any constraints.
Copy()	Copies both the structure and data for this DataSet.
GetChanges()	Returns a copy of the DataSet containing all changes made to it since it was last loaded or since AcceptChanges() was called.
GetChildRelations()	Returns the collection of child relations that belong to a specified table.
GetParentRelations()	Gets the collection of parent relations that belong to a specified table.
HasChanges()	Overloaded. Gets a value indicating whether the DataSet has changes, including new, deleted, or modified rows.
Merge()	Overloaded. Merges this DataSet with a specified DataSet.

ReadXml()
ReadXmlSchema()
	Allow you to read XML data from a valid stream (file based, memory based, or network based) into the DataSet.

RejectChanges()	Rolls back all the changes made to this DataSet since it was created or the last time DataSet.AcceptChanges was called.

WriteXml()
WriteXmlSchema()
	Allow you to write out the contents of a DataSet into a valid stream.

Table 22-11. *Properties of the* DataColumn

Properties	Meaning in Life
AllowDBNull	This property is used to indicate if a row can specify null values in this column. The default value is true.
AutoIncrement AutoIncrementSeed AutoIncrementStep	These properties are used to configure the autoincrement behavior for a given column. This can be helpful when you wish to ensure unique values in a given DataColumn (such as a primary key). By default, a DataColumn does not support autoincrement behavior
Caption	This property gets or sets the caption to be displayed for this column (e.g., what the end user sees in a DataGridView).
ColumnMapping	This property determines how a DataColumn is represented when a DataSet is saved as an XML document using the DataSet.WriteXml() method.
ColumnName	This property gets or sets the name of the column in the Columns collection (meaning how it is represented internally by the DataTable). If you do not set the ColumnName explicitly, the default values are Column with (*n*+1) numerical suffixes (i.e., Column1, Column2, Column3, etc.).
DataType	This property defines the data type (Boolean, string, float, etc.) stored in the column.
DefaultValue	This property gets or sets the default value assigned to this column when inserting new rows. This is used if not otherwise specified.

Working with Data Adapters
Now that you understand the ins and outs of manipulating ADO.NET DataSets, let's turn our attention to the topic of data adapters. Recall that data adapter objects are used to fill a DataSet with DataTable objects and send modified DataTables back to the database for processing. Table 22-15 documents the core members of the DbDataAdapter base class

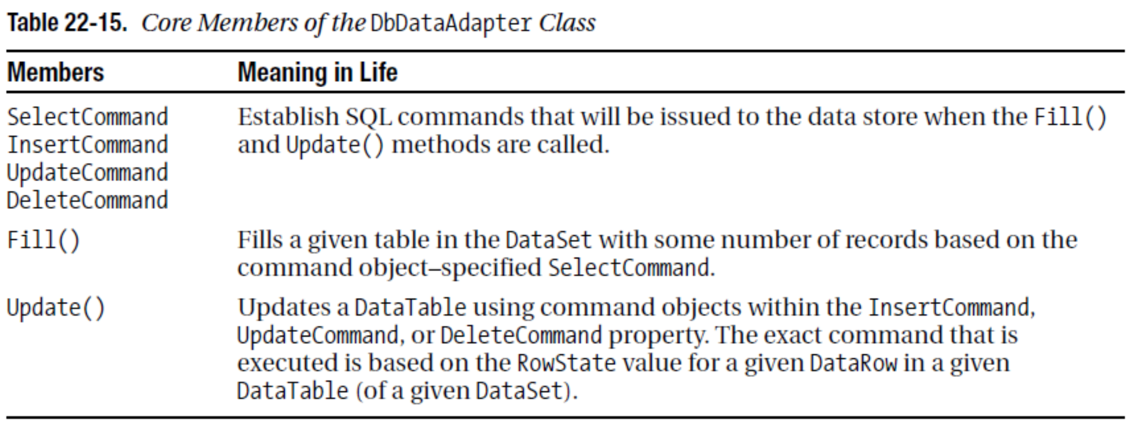


Table 22-15. *Core Members of the* DbDataAdapter *Class*

Members	Meaning in Life
SelectCommand InsertCommand UpdateCommand DeleteCommand	Establish SQL commands that will be issued to the data store when the Fill() and Update() methods are called.
Fill()	Fills a given table in the DataSet with some number of records based on the command object–specified SelectCommand.
Update()	Updates a DataTable using command objects within the InsertCommand, UpdateCommand, or DeleteCommand property. The exact command that is executed is based on the RowState value for a given DataRow in a given DataTable (of a given DataSet).

. ADO.NET is a new data access technology developed with the disconnected *n*-tier application firmly in mind. The System.Data namespace contains most of the core types you need to programmatically interact with rows, columns, tables, and views. As you have seen, the .NET platform ships with numerous data providers that allow you to leverage the connected and disconnected layers of ADO.NET.
Using connection objects, command objects, and data reader objects of the connected layer, you are able to select, update, insert, and delete records. As you have seen, command objects support an internal parameter collection, which can be used to add some type safety to your SQL queries and are quite helpful when triggering stored procedures.
The centerpiece of the disconnected layer is the DataSet. This type is an in-memory representation of any number of tables and any number of optional interrelationships, constraints, and expressions. The beauty of establishing relations on your local tables is that you are able to programmatically navigate between them while disconnected from the remote data store. You also examined the role of the data adapter in this chapter. Using this type (and the related SelectCommand, InsertCommand, UpdateCommand, and DeleteCommand properties), the adapter can resolve changes in the DataSet with the original data store. Also, you learned about the connected layer of ADO.NET and came to understand the role of data reader types.

THANKS