C# PROGRAMMING AND .NET

Subject Code: 10IS761/10CS761 **I.A. Marks : 25**
Hours/Week : 04 **Exam Hours: 03**
Total Hours : 52 **Exam Marks: 100**

PART – A

UNIT – 1 **6 Hours**

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

UNIT – 2 **6 Hours**

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

UNIT – 3 **8 Hours**

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

UNIT – 4 **6 Hours**

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

PART – B

UNIT – 5 **6 Hours**

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

UNIT – 6 **6 Hours**

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

UNIT – 7 **8 Hours**

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

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

UNIT – 8 **6 Hours**

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

Text Books:

1. Andrew Troelsen: Pro C# with .NET 3.0, Special Edition, Dream tech Press, India, 2007. Chapters: 1 to 11 (up to pp.369)

TABLE OF CONTENTS

UNIT 1: THE PHILOSOPHY OF .NET

UNDERSTANDING THE PREVIOUS STATE OF AFFAIRS

Life as a C API Programmer

• C is a very complex language because C programmers were forced to deal with
  → manual memory management
  → ugly pointer arithmetic
  → ugly syntactic-constructs
• C is a structured language and so lacks the benefits provided by object-oriented approach.
• When you combine thousands of global-functions & data-types defined by the Win32 API to a difficult language like C, bugs will increase rapidly.

Life as a C++ Programmer

• Though C++ provides OOPs concepts like encapsulation, inheritance & polymorphism, C++ programming remains a difficult & error-prone experience, given its historical roots in C.

Life as a Visual Basic 6.0 Programmer

• VB6 is popular due to its ability to build
  → complex user interfaces
  → code libraries &
  → simpler data access logic
• The major downfall of VB is that it is not a fully object-oriented language, but rather "object-aware". For example,
  → It does not allow to establish "is-a" relationships between types
  → It does not support the ability to build multi-threaded applications
  → It has no support for parameterized class construction

Life as a Java Programmer

• Java has greater strength because of its platform independence nature.
• One potential problem using Java typically means that "you must use Java front-to-back during development cycle".
• Pure java is simply not appropriate for graphically intensive applications due to its slow execution speed.
• Java provides a limited ability to access non-Java APIs and hence, there is little support for true cross-language integration.

Life as a Windows-DNA Programmer

• The popularity of Web-applications is ever expanding. Sadly, building a web-application using COM-based Windows-DNA is quite complex.
• Some of this complexity is because Windows-DNA requires use of numerous languages like ASP, HTML, XML, JavaScript, VBScript, ADO etc.
• Many of these technologies are completely unrelated from a syntactic-point of view.

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

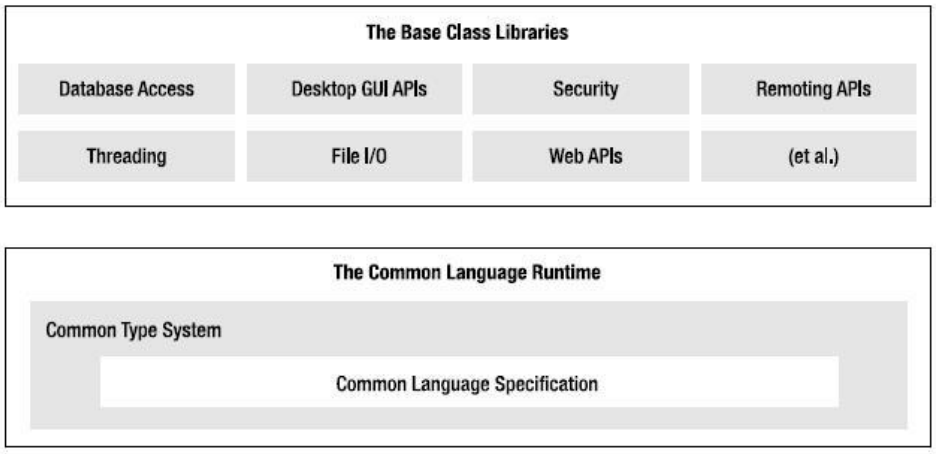


Figure 1-1. The CLR, CTS, CLS, and base class library relationship

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

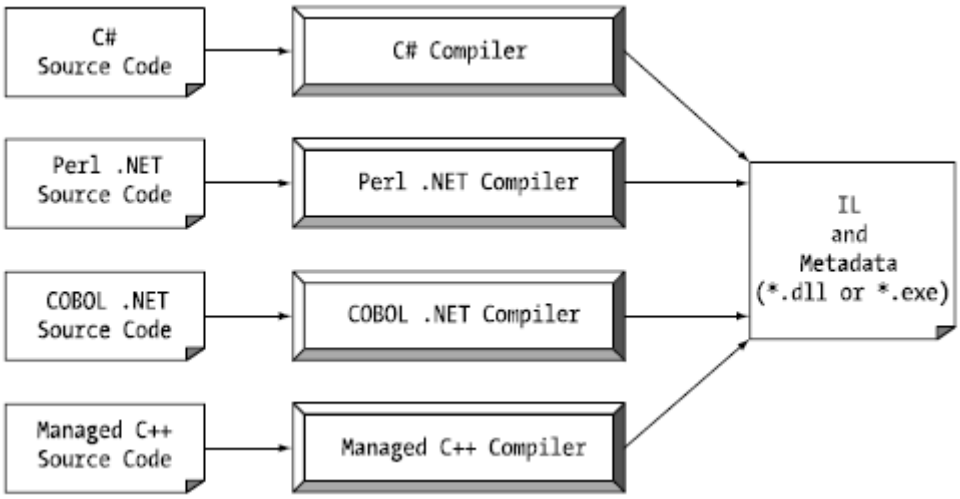


Figure 1-2: All .NET-aware compilers emit IL instructions and metadata

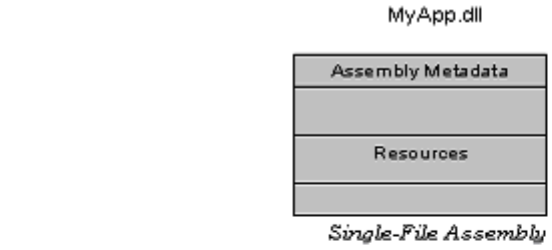
Managed code & unmanaged code

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

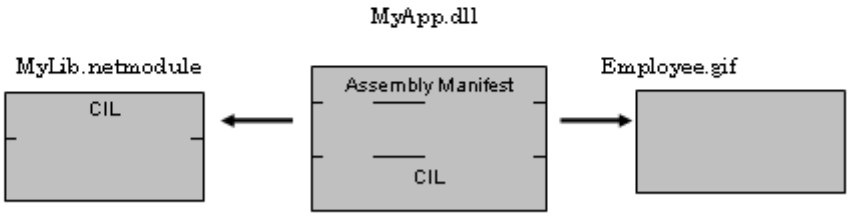
• Conversely, code that cannot be directly hosted by the .NET runtime is termed unmanaged-code.

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 multifile assembly, one of these modules (termed the primary module) must contain the assembly manifest (and possibly CIL instructions and metadata for various types).
• The other related modules contain a module level manifest, CIL, and type metadata.
• 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.
• Multifile 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: Multifile 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.

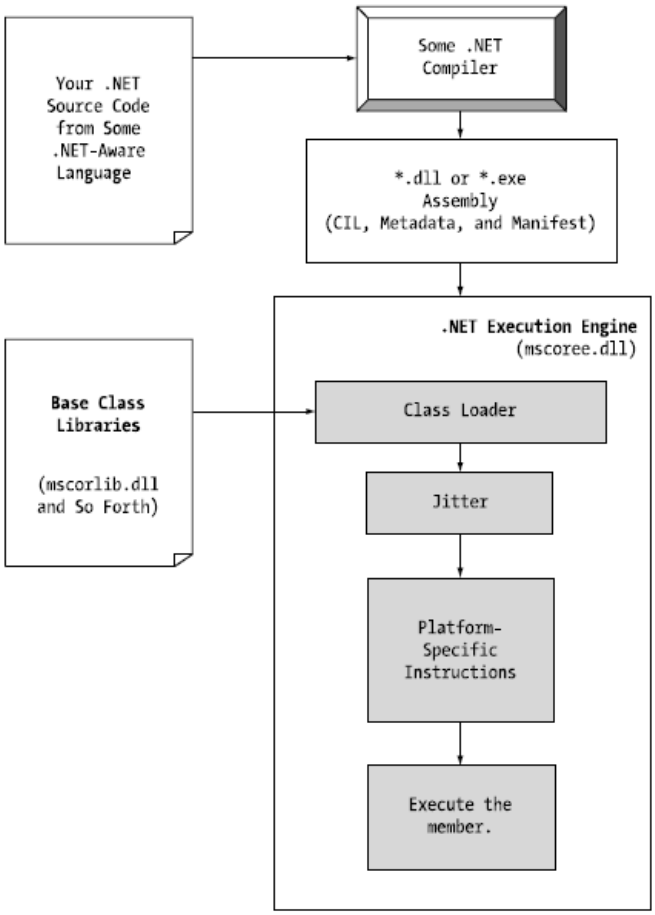


Figure 1-3: mscoree.dll in action

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).

EXERCISES

1. Briefly discuss state of affairs that eventually led to .NET platform. (8)
2. Explain .NET solution. (4)
3. What are the key features of C#? (4)
4. With a neat diagram, explain basic building block of .NET framework. (8)
5. What do you mean by Base class library? Explain. (2)
6. Explain the concept of .NET binaries. (6)
7. Bring out important differences b/w single and multifile assemblies. (4)
8. Explain the role of CIL and the benefits of CIL. (4)
9. Explain the role of JIT compiler. (6)
10. What are basic CTS data types? Explain. (8)
11. Explain common type system in detail. (6)
12. Explain with a neat diagram, the workflow that takes place between your source code, a given .NET complier and the .NET execution engine. (8)
13. What are namespaces? List and explain the purpose of at least five namespaces. (6)

UNIT 2: BUILDING C# APPLICATIONS

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 TestApp
{
        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.

OPTIONS OF THE C# COMMAND LINE COMPILER

/target
This is used to specify format of output-file.
/out
This is used to specify name of output-file.
/nostdlib
This is used to prevent automatic importing of core .NET library 'mscorlib.dll'.
/reference
This is used to reference an external assembly.
@
This allows to specify a response-file used during compilation.
/noconfig
This is used to prevent use of response-files during the current compilation.
/bugreport
This is used to build text-based bug-reports for the current compilation.
/main
This is used to specify which Main() method to use as the program's entry point, if multiple Main() methods have been defined in the current *.cs file-set.
/? Or /help
This prints out list of all command-line flags of compiler.
/addmodule
This is used to specify modules to be added into a multi-file assembly.
/nologo
This is used to suppress banner-information when compiling the file.
/debug
This forces compiler to emit debugging-information.
/define
This is used to define pre-processor symbols.
/optimize
This is used to enable/disable optimizations.
/warn
This is used to set warning-level for the compilation cycle.
/doc
This is used to construct an XML documentation-file.
/filealign
This is used to specify size of sections in output-file.
/fullpaths
This is used to specify absolute path to the file in compiler output.
/incremental
This is used to enable incremental compilation of source-code files.
/lib
This is used to specify location of assemblies referenced via /reference.
/linkresource
This is used to create a link to a managed-resource.
/baseaddress
This is used to specify preferred base-address at which to load a *.dll
/checked
This is used to specify the code-page to use for all source-code files in the compilation.

THE COMMAND LINE DEBUGGER (cordbg.exe)

• This tool provides various options that allow running your assemblies under debug-mode.
• You may view options by specifying the -? Flag:

cordbg -?

• List of cordbg.exe flags are:

b[reak] :Set or display current breakpoints
del[ete] :Remove one or more breakpoints
ex[it] :exit the debugger
g[o] :continue debugging the current process until hitting next breakpoint
si :step into the next line
o[ut] :step out of the current function
so :step over of the next line
p[rint] :print all loaded variables

Debugging at the Command Line

• Firstly generate symbolic debugging symbols for your current application by specifying the /debug flag of csc.exe:

csc @testapp.rsp /debug

• The above command-set generates a new file named *testapp.pdb*. If you do not have an associated *.pdb file, it is still possible to make use of cordbg.exe, however, you will not be able to view your C# source code during the process.
• Once you have a valid *.pdb file, open a session with cordbg.exe by specifying your assembly as a command line argument(the *.pdb file will be loaded automatically):

cordbg.exe testapp.exe

• At this point, you are in debugging mode, and may apply any number of cordbg.exe flags at the (cordbg)" command prompt.

CORE PROJECT WORKSPACE TYPES IN VS.NET IDE

Windows Application
This represents a windows forms application.
Class Library
This allows you to build a single file assembly.
Windows Control Library
This allows you to build a single file assembly that contains custom windows forms controls.
ASP.NET Web Application
This is selected when you want to build an ASP.NET web application.
ASP.NET Web Service
This allows you to build a .NET web services. A web service is a block of code, reachable using HTTP requests.
Web Control Library
This allows you to build customized web controls. These GUI widgets are responsible for emitting HTML to a requesting browser.
Windows Services
This allows you to build NT/2000 services. These are background worker applications that are launched during the OS boot process.

THE STRUCTURE OF A VS.NET CONSOLE APPLICATION

\bin\Debug
This folder contains the debug version of your compiled assembly. If you configure a release build, a new folder (\bin\Release) will be generated that contains a copy of your assembly, stripped of any debugging information.
\obj*
This folder consists of numerous subfolders used by VS.NET during the compilation process.
App.ico
This file is used to specify the icon for the current program.
AssemblyInfo.cs
This file allows you to establish assembly-level attributes for your current project.
Class1.cs
This file is your initial class file.
***.csproj**
This file represents a C# project that is loaded into a given solution.
***.sln**
This file represents the current VS.NET solution.

RUNNING VERSUS DEBUGGING

• When you run an application, you are instructing VS.NET to ignore all breakpoints to automatically prompt for a keystroke before terminating the current console window.
• On the other hand, if you debug a project that does not have any breakpoints set, the console application terminates so quickly that you will be unable to view the output.
• To ensure that the command window is alive regardless of the presence of a given breakpoint, add the following code at the end of the Main() method:

```
static void Main(string[] args)
{
        Console.ReadLine();      //keep console window up until user hits return.
}
```

EXAMINING THE SOLUTION EXPLORER WINDOW

• A solution is a collection of one or more projects.
• Each project contains number of source code files, external references and resources that constitute the application as a whole.
• Regardless of which project workspace type you create, *.sln file can be opened using VS.NET to load each project in the workspace (Figure: 2.11).
• The solution explorer window provides a class view tab, which shows the object-oriented view of your project (Figure: 2.12).

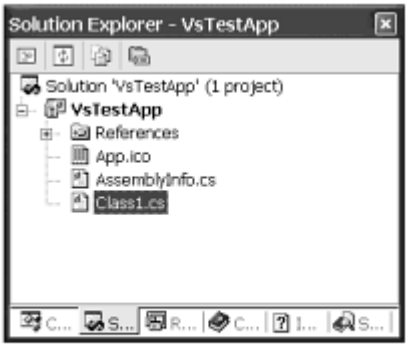


Figure 2-11. The Solution Explorer

Figure 2-12. Class View

EXAMINING THE SERVER EXPLORER WINDOW

• This window can be accessed using the View menu (Figure: 2.20)

• This window can be thought of as the command center of a distributed application you may be building.

• Using this, you can

→ attach to and manipulate local and remote databases

→ examine the contents of a given message queue &

→ obtain general machine-wide information

Figure 2-20. The Server Explorer window

DOCUMENTING YOUR SOURCE CODE VIA XML

- Q: Why use XML to document type definitions rather than HTML?
  Ans: 1. Given that XML separates the definition of data from the presentation of that data, you can apply any number of XML transformations to the raw XML.
  2. You can also programmatically read & modify the raw XML using types defined in the System.Xml.dll assembly.
- When you want to document the types, you have to make use of special comment syntax, /// (rather than C++ style // or C based comment syntax /* . . . */)
- **List of Stock XML Tags:**

<c>
This indicates that text within a description should be marked as code.
<code>
This indicates multiple lines should be marked as code.
<example>
This is used to mock-up a code-example for the item you are describing.
<exception>
This is used to document which exceptions a given class may throw.
<list>
This is used to insert a list into the documentation-file.
<param>
This describes a given parameter.
<paramref>
This associates a given XML tag with a specific parameter.
<permission>
This is used to document access-permissions for a member.
<remarks>
This is used to build a description for a given member.
<returns>
This documents the return value of the member.
<see>
This is used to cross-reference related items.
<summary>
This documents the "executive summary" for a given item.
<value>
This documents a given property.

- **List of XML Format Characters**

N :This denotes a namespace.
T :This represents a type(e.g. ,class, struct, enum, interface)
F :This represents a field.
P :This represents type properties.
M :This represents method including constructors and overloaded operators.
E :This denotes an event.
! :This represents an error string that provides information about the error. The compiler generates error information for links that cannot be resolved.

```
/// <summary>
///  This is a simple Car that illustrates
///  working with XML style documentation.
/// </summary>
public class Car
{
    /// <summary>
    /// Do you have a sunroof?
    /// </summary>
    private bool hasSunroof = false;

    /// <summary>
    /// The ctor lets you set the sunroofedness.
    /// </summary>
    /// <param name="hasSunroof"> </param>
    public Car(bool hasSunroof)
    {
        this.hasSunroof = hasSunroof;
    }

  /// <summary>
  /// This method allows you to open your sunroof.
  /// </summary>
  /// <param name="state"> </param>
  public void OpenSunroof(bool state)
  {
      if(state == true && hasSunroof == true)
          Console.WriteLine("Put sunscreen on that bald head!");
      else
          Console.WriteLine("Sorry...you don't have a sunroof.");
  }
  /// <summary>
  /// Entry point to application.
  /// </summary>
  public static void Main()
  {
      Car c = new Car(true);
      c.OpenSunroof(true);
  }
```

Figure 2-28. XmlCarDoc.xml

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.

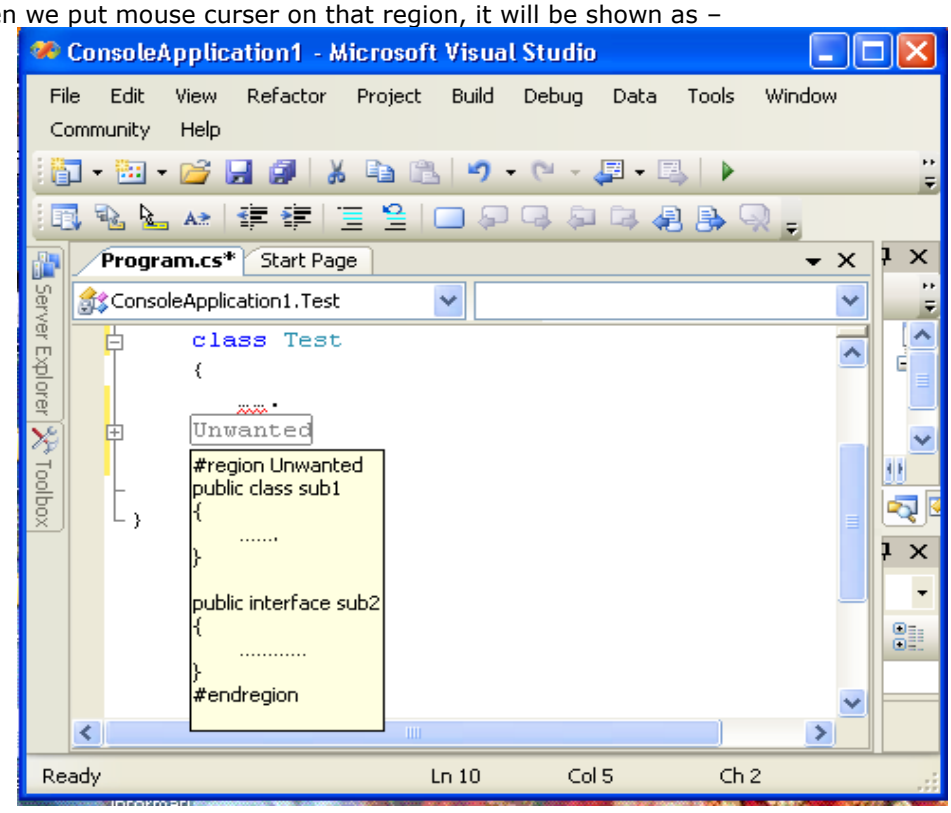
SPECIFYING CODE REGIONS

• 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 –



CONDITIONAL CODE COMPILATION

• The preprocessor directives #if, #elif, #else and #endif allows to conditionally compile a block of code based on predefined symbols. Consider the following example –

```
#define MAX  100
using System;

class Test
{
      public static void Main()
      {
        #if(MAX)
            Console.WriteLine("MAX is defined");
        #else
            Console.WriteLine("MAX is not defined");
        #endif

      }
}
```

• The output will be:

```
MAX is defined
```

ISSUING WARNINGS AND ERRORS

• The *#warning* and *#error* directives allow instructing the C# compiler to generate a warning or error.

• For example, we wish to issue a warning when a symbol is defined in the current project.

• This is helpful when a big project is handled by many people.

• Consider the following code:

```
1    #define MAX 100
2    using System;
3    class Test
4    {
5          public static void Main()
6          {
7              #if(MAX)
8              #warning MAX is defined by me!!!
9                ……..
10             #endif
11         }
12   }
```

• When we compile, the program will give warning message –

```
Test.cs(8,21): warning CS1030: #warning: 'MAX is defined by me!!!'
```

ALTERING LINE NUMBERS

• The directive #line allows altering the compiler's recognition of #line numbers during its recording of compilation warnings and errors.

• To reset the default line numbering, we can specify *default* tag. Consider the following code:

```
1    #define MAX 100
2    using System;
3    class Test
4    {
5          public static void Main()
6          {
7              #line 300              //next line will take the number 300
8              #warning MAX is defined by me!!!
9                    ……..
10             #line default    // default numbering will continue
11         }
12   }
```

• When we compile, the program will give warning message –

```
Test.cs(300,21): warning CS1030: #warning: 'MAX is defined by me!!!'
```

• Note that the line number appeared as 300, though the actual line was 7.

THE SYSTEM.ENVIRONMENT CLASS

• This class can be used to obtain a number of details regarding the context of the OS hosting your application using various static members. For example, consider the following code:

```
using System;
class PlatformSpy
{
        public static int Main(strng[] args)
        {
                Console.WriteLine("Current OS:{0}",Environment.OSVersion);
                Console.WriteLine("Current directory :{0}",Environment.CurrentDirectory);

                string[] drives=Environment.GetLogicalDrives();
                for(int i=0;i<drives.Length;i++)
                        Console.WriteLine("Drive {0} : {1}",i,drives[i]);

                Console.WriteLine("Current version of .NET :{0}",Environment.Version);
                return 0;
        }
}
```

• The Output will be -

```
Current OS: Microsoft Windows NT 5.1.2600 Service Pack 3
Current directory: C:\Program Files\Microsoft Visual Studio 8\SDK\v2.0
Drive 0: A:\
Drive 1: C:\
Drive 2: D:\
Drive 3: E:\
Drive 4: F:\
Current version of .NET: 2.0.50727.3082
```

EXERCISES:

1. Explain the role of csc.exe. (6)
2. List and explain the basic output options available with C# complier. (6)
3. Explain the following, with respect to compilation of C# program: (6)
   i) referencing external assemblies
   ii) compiling multiple source files
4. What are response files? Explain with an example. (4)
5. How do you generate bug reports? Illustrate with an example. (6)
6. What are C# preprocessor directives? Explain. (6)
7. Explain the following aspects of VS.NET IDE: (8)
   i) Solution Explorer
   ii) Running and debugging
   iii) Documenting code via XML
8. Explain the use of System.Environment class with a program. (4)

UNIT 3: C# LANGUAGE FUNDAMENTALS

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.

CREATING OBJECTS: CONSTRUCTOR BASICS

• Consider the following code:

```
using System;
class HelloClass
{
        //default constructor always assigns state data to default values
    public HelloClass()
    {
        Console.WriteLine("default constructor called");
    }

     //custom constructor assigns state data to a known value
    public HelloClass(int x, int y)
    {
        Console.WriteLine("custom constructor called");
        inX=x;
        inY=y;
    }

    public int inX,inY;                                    //some public state data

    public static int Main(string[] args)
    {
        HelloClass h1=new HelloClass();          //trigger default constructor
        Console.WriteLine("h1.intX={0} h1.intY={1}",h1.inX,h1.inY)

        HelloClass h2;
        h2=new HelloClass(100,255);          //trigger custom constructor
        Console.WriteLine("h2.intX={0} h2.intY={1}",h2.inX,h2.inY);

        return 0;
    }
}
```

• A *class* is a definition of a user-defined type(UDT).
• The class can be regarded as a blueprint for variables of this type.
• An *object* can be described as a given instance of a particular class.
• The "new" keyword is responsible for
→ allocating correct number of bytes for the specified class &
→ acquiring sufficient memory from heap.
• Object-variables are actually a reference to the object in memory, not the actual object itself. (Thus, h1 and h2, each reference a distinct 'HelloClass' object allocated on the heap).
• By default, every class is gifted with a default-constructor, which we are free to redefine if needed.
• Default constructor ensures that all member-data is set to an appropriate default values.
• Constructors are named identical to the class they are constructing and do not take a return value.
• C# programmers never explicitly destroy an object.
• The *.NET garbage collector* frees the allocated memory automatically, and therefore C# does not support a "delete" keyword.

DEFAULT ASSIGNMENTS AND VARIABLE SCOPE

• Consider the following code:

```
class DefaultValueTester
{
        public int theInt;          //assigned to 0
        public long theLong;        //assigned to 0.0
        public char theChar;        //assigned to 0"
        public bool theBool;        //assigned to false
        public string theStr;       //assigned to null
        public object theObj;       //assigned to <undefined value>

        public static int Main(string[] args)
        {
                DefaultValueTester d=new DefaultValueTester();
                 return 0;
        }
}
```

• All .NET data types have a default value.

• When we create custom types (i.e. class, struct), all member-variables are automatically assigned to their appropriate default value.

• When we define variable within a method-scope, we must assign an initial value before we use them, as they do not receive a default assignment.

• If the variable is functioning as an "output" parameter, the variable does not need to be assigned an initial value.

• Methods that define *output parameters* assign incoming variables within their function scope before the caller makes direct use of them.

BASIC INPUT AND OUTPUT WITH THE CONSOLE CLASS

• *Console* class encapsulates input, output and error stream manipulations.

• Important methods of Console class:

i. Read() is used to capture a single character from the input-stream.
ii. ReadLine() allows to receive information from input-stream up until carriage-return.
iii. Write() pumps text to the output-stream without a carriage-return.
iv. WriteLine() pumps a text string (including a carriage-return) to the output-stream.

• Consider the following code:

```
using System;
class BasicIO
{
        public static void Main(string[] args)
        {
                string s, a;

                Console.Write("enter your name");
                s=Console.ReadLine();
                Console.WriteLine("hello {0}",s);

                Console.Write("enter your age");
                a=Console.ReadLine();
                Console.WriteLine("you are {0} years old",a);
        }
}
```

Output:

```
enter your name: john
hello john
enter your age:25
you are 25 years old
```

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

FORMATTING TEXTUAL OUTPUT

• Consider the following code:

```
using System;
class BasicIO
{
        public static void Main()
        {
                int i=77;
                float f=7.62.;
                string s="hello";

                Console.WriteLine("Integer is: {0}\n Float is :{1}\n String is :{2}", i, f, s);
        }
}
```

Output:

```
Integer is: 77
Float is :7.62
String is :hello
```

• The first parameter to WriteLine() represents a format-string that contains optional *placeholders* designated by {0},{1},{2}.

• The remaining parameters to WriteLine() are the values to be inserted into the respective placeholders.

• WriteLine() has been overloaded to allow us to specify placeholder values as an array of objects. For example, consider the following lines of code.

```
object[] stuff={"hello",20.9,"there","1986"};
Console.WriteLine("the stuff: {0},{1},{2},{3}",stuff);
```

• A given placeholder can be repeated within a given string. For example, consider the following line of code.

```
Console.WriteLine("{0}number{0}number{0}",9);
```

STRING FORMATTING FLAGS

Flags	Meaning
C or c	Used to format currency. By default, the flag will prefix the local cultural symbol ($ for US English). However, this can be changed using System.Globalization.NumberFormatInfo object.
D or d	Used to format decimal numbers. This flag may also specify the minimum number of digits used to pad the value.
E or e	Exponential notation.
F or f	Fixed point formatting.
G or g	G stands for general. Used to format a number to fixed or exponential format.
N or n	Basic numerical formatting with commas.
X or x	Hexadecimal formatting.

• Consider the following code:

```
public static void Main(string[] args)
{
        Console.WriteLine("C format:{0:C}",99987.99);          //prints $99,987.99
        Console.WriteLine("D9 format:{0:D9}",99987);           //prints 00009.9987
        Console.WriteLine("E format:{0:E}",99987.76543);       //prints 9.9987E+004
        Console.WriteLine("F3 format:{0:F3}",99999.9999);      //prints 100000.000
        Console.WriteLine("N format:{0:N}",99987);             //prints 99,987.00
        Console.WriteLine("X format:{0:X}",99987);             //prints 1869F
}
```

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

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's again 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's again 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;              // unboxing successful
string 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
        }
}
```

Output:

```
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
```

- From the above example, we can observe that for *params* parameter, we can pass an array or individual elements.
- We can use *params* even when the parameters to be passed are of different types.
- Consider the following program:

```
using System;
class Test
{
        public static void MyFun(params object[] arr)
        {
                for(int i=0; i<arr.Length; i++)
                {
                        if(arr[i] is Int32)
                            Console.WriteLine("{0} is an integer", arr[i]);
                        else if(arr[i] is string)
                                Console.WriteLine("{0} is a string", arr[i]);
                             else if(arr[i] is bool)
                                        Console.WriteLine("{0} is a boolean",arr[i]);
                }
        }

        public static void Main()
        {
                int x=5;
                string s="hello";
                bool b=true;

                MyFun(b, x, s);
        }
}
```

Output:

```
True is a Boolean
5 is an integer
hello is a string
```

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.

EXERCISES

1) Explain the anatomy of a Basic C# Class. (6)
2) Explain the 3 methods of processing command line parameters. (6)
3) With example, explain objects & constructors. (6)
4) With example, explain default assignments and variable scope. (4)
5) What are basic input & output functions in Console class? Explain with example. (6)
6) Write a note on formatting textual input. (4)
7) List out & explain the string formatting flags with example. (4)
8) Explain value types and reference types with example for each. (6)
9) Compare value types vs. reference types (4)
10) With example, explain value types containing reference types. (6)
11) What is boxing and unboxing? Explain with example. (6)
12) Explain various method parameter modifiers with example for each. (8)
13) With example, explain passing reference types by value and reference. (6)

THE MASTER NODE: SYSTEM.OBJECT

- In .NET, every data type is derived from a common base class: *System.Object*.
- The Object class defines a common set of members supported by every type in the .NET framework.
- When we create a class, it is implicitly derived from System.Object.
- For example, the following declaration is common way to use.

```
class Test                                  class Test : System.Object
{                                           {
 ...          But, internally, it means that  ...
}                                           }
```

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

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

Instance Method of Object Class	Meaning
Equals()	By default, this method returns true only if the items being compared refer to the exact same item in memory. Thus, Equals() is used to compare object references, not the state of the object. Typically, this method can be overridden to return true only if the objects being compared have the same internal state values.
GetHashCode()	This method returns an integer that identifies a specific object in memory.
GetType()	This method returns a System.Type object that fully describes the details of the current item.
ToString()	This method returns a string representation of a given object, using the *namespace.typename* format (i.e., fully qualified name). If the type has not been defined within a namespace, *typename* alone is returned. Typically, this method can be overridden by a subclass to return a tokenized string of name/value pairs
Finalize()	This method is invoked by the .NET runtime when an object is to be removed from the heap (during garbage collection).
MemberwiseClone()	This method is used to return a new object that is a member-by-member copy of the current object. Thus, if the object contains references to other objects, the *references* to these types are copied (i.e., it achieves a shallow copy). If the object contains value types, full copies of the values are achieved (i.e., it achieves a deep copy).

THE DEFAULT BEHAVIOR OF SYSTEM.OBJECT

• Consider the following code:

```
using System;
class Person
{
        public string Name, SSN;
        public byte age;

        public Person(string n, string s, byte a)
        {
                Name = n;
                SSN = s;
                age = a;
        }

        public Person(){ }

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

                Person p1 = new Person("Ram",  "111-11-1111", 20);
                Console.WriteLine("p1.ToString: {0}", p1.ToString());
                Console.WriteLine("p1.GetHashCode: {0}", p1.GetHashCode());
                Console.WriteLine("p1's base class: {0}", p1.GetType().BaseType);

                Person p2 = p1;
                object o = p2;

                if(o.Equals(p1) && p2.Equals(o))
                        Console.WriteLine("p1, p2 and o are same objects!");
        }
}
```

Output:

```
***** Working with Object *****
p1.ToString: Person
p1.GetHashCode: 58225482
p1's base class: System.Object
p1, p2 and o are same objects!
```

• In the above program, the default implementation of ToString() simply returns the fully qualified name of the type.

• GetType() retrieves a System.Type object, which defines a property named.

• Here, new Person object p1 is referencing memory in the heap.

• We are assigning p2 to p1. Therefore, p1 and p2 are both pointing to the same object in memory.

• Similarly the variable o (of type object) also refers to the same memory. (Thus, when we compare p1, p2 and o, it says that all are same).

OVERRIDING SOME DEFAULT BEHAVIORS OF SYSTEM.OBJECT

• In many of the programs, we may want to override some of the behaviors of System.Object.

• O*verriding* is the process of redefining the behavior of an inherited *virtual* member in a derived class.

• We have seen that System.Object class has some virtual methods like ToString(), Equals() etc. These can be overridden by the programmer.

OVERRIDING TOSTRING()

• Consider the following code:

```
using System;
using System.Text;
class Person
{
        public string Name, SSN;
        public byte age;

        public Person(string n, string s, byte a)
        {
                Name = n;
                SSN = s;
                age = a;
        }

        public Person(){ }

        // Overriding System.Object.ToString()
        public override string ToString()
        {
                StringBuilder sb = new StringBuilder();
                sb.AppendFormat("[Name={0}", this. Name);
                sb.AppendFormat(" SSN={0}", this.SSN);
                sb.AppendFormat(" Age={0}]", this.age);
                return sb.ToString();
        }

        public static void Main()
        {
                Person p1 = new Person("Ram", "11-12", 25);
                Console.WriteLine("p1 is {0}", p1.ToString());
        }
}
```

Output:

```
p1 is [Name=Ram SSN=11-12 Age=25]
```

• In the above example, we have overridden ToString() method to display the contents of the object in the form of tuple.

• The System.Text.StringBuilder is class which allows access to the buffer of character data and it is a more efficient alternative to C# string concatenation.

OVERRIDING EQUALS()

• By default, System.Object.Equals() returns true only if the two references being compared are referencing same object in memory.

• But in many situations, we are more interested if the two objects have the same content. Consider an example:

```
using System;
class Person
{
        public string Name, SSN;
        public byte age;

        public Person(string n, string s, byte a)
        {
                Name = n;
                SSN = s;
                age = a;
        }

        public Person(){ }

        public override bool Equals(object ob)
        {
           if (ob != null && ob is Person)
           {
              Person p = (Person)ob;

              if (p.Name == this.Name && p.SSN == this.SSN && p.age == this.age)
                    return true;
           }
           return false;
        }

        public static void Main()
        {
                Person p1 = new Person("Ram", "11-12", 25);
                Person p2 = new Person("John", "11-10", 20);
                Person p3 = new Person("Ram", "11-12", 25);
                Person p4 = p2;
                if(p1.Equals(p2))
                     Console.WriteLine("p1 and p2 are same");
                else
                     Console.WriteLine("p1 and p2 are not same");

                if(p1.Equals(p3))
                    Console.WriteLine("p1 and p3 are same");
                else
                    Console.WriteLine("p1 and p3 are not same");

                if(p2.Equals(p4))                    //compares based on content, not on reference
                    Console.WriteLine("p4 and p2 are same");
                else
                    Console.WriteLine("p4 and p2 are not same");
        }
}
```

Output:

```
p1 and p2 are not same
p1 and p3 are same
p4 and p2 are same
```

• While overriding the Equals() method, first we are checking whether the passed object is of class *Person* or not.

• Also, we need to check whether the object has been allocated memory or it is having *null*. • Note that Equals() method takes the parameter of type *object*. Thus, we need to type-cast it to *Person* type before using it.

• When we override Equals(), we need to override GetHashCode() too.

OVERRIDING SYSTEM.OBJECT.GETHASHCODE()

• The GetHashCode() method returns a numerical value used to identify an object in the memory.

• The default implementation of the GetHashCode() method does not guarantee unique return values for different objects. (Thus, if we have two Person objects that have an identical name, SSN, and age, obtain the same hash code).

• When a class overrides the Equals() method, best practices dictate that we should also override System.Object.GetHashCode(). If we fail to do so, we are issued a compiler warning.

• Consider the following code:

```
using System;
class Person
{
  public string Name, SSN;
  public byte age;

  public Person(string n, string s, byte a)
  {
    Name = n;
    SSN = s;
    age = a;
  }

  public Person(){ }

  public override int GetHashCode()
  {
    return SSN.GetHashCode();
  }

  public static void Main()
  {
    Person p1 = new Person("Ram", "11-12", 25);
    Person p2 = new Person("John", "11-10", 20);
    Person p3 = new Person("Ram", "11-12", 25);
    Person p4 = p2;

    if(p1.Equals(p3))                                    //comparison based on reference
        Console.WriteLine("p1 and p3 are same");
    else
        Console.WriteLine("p1 and p3 are not same");

    if(p1.GetHashCode()==p3.GetHashCode())          //comparison based on SSN
        Console.WriteLine("p1 and p3 are same");
    else
        Console.WriteLine("p1 and p3 are not same");
  }
}
```

Output:

```
p1 and p3 are not same
p1 and p3 are same
```

ITERATION CONSTRUCTS

• Similar to any other programming language, C# provides following iteration constructs:

→ for Loop
→ foreach/in Loop
→ while Loop
→ do/while Loop

FOR LOOP

• This loop will allow to repeat a set of statements for fixed number of times.

• We can use any type of complex terminating conditions, incrementation/decrementation, continue, break, goto etc. while using for loop.

FOREACH/IN LOOP

• This loop is used to iterate over all items within an array, without the need to test for the array's upper limit. For example:

```
using System;
class Program
{
   static void Main()
   {
            // ... loop with the for keyword.
             for(int i=0; i<4;i++)
                   Console.WriteLine("{0} ", i);        //0 to 3 will be printed.

             string[] pets = { "dog", "cat", "bird" };
             // ...now loop with the foreach keyword.
             foreach (string value in pets)
             {
                      Console.WriteLine(value);
             }
   }
}
```

Output:

```
0  1  2  3
dog
cat
bird
```

• Apart from iterating over simple arrays, foreach loop can be used to iterate over system supplied or user-defined collections.

WHILE AND DO/WHILE LOOP

• When we don't know the exact number of times a set of statements to be executed, we will go for *while* loop. That is, a set of statements will be executed till a condition remains true. For example:

```
using System;
class Program
{
   static void Main()
   {
            // Continue in while-loop until index is equal to 3.
            int i = 0;
            while (i < 3)
            {
                     Console.Write("C# Language ");
                      // Write the index to the screen.
                     Console.WriteLine(i);
                      // Increment the variable.
                     i++;
            }
   }
}
```

Output

```
C# Language 0
C# Language 1
C# Language 2
```

• Some times, we need a set of statements to be executed at least once, irrespective of the condition. Then we can go for *do/while* loop. For example:

```
string opt;
do
{
        Console.Write("Do you want to continue?(Yes/No): ");
        opt=Console.ReadLine();
}while(opt!="Yes");
```

CONTROL FLOW CONSTRUCTS

• There are two control flow constructs in C# viz. *if/else* and *switch/case*.
• *if/else* works only on boolean expressions. So, we cannot use the values 0, 1 etc. within *if* as we do in C/C++.
• Switch/case allows us to handle program flow based on a predefined set of choices.
• A condition for a value "not equal to zero" will not work in C#.
• The following two tables list out the relational and logical operators respectively.

Relational Operator	Meaning
==	To check equality of two operands
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

Logical Operator	Meaning
&&	Logical AND
||	Logical OR
!	Logical NOT

• For example:

```
class Selections
{
        public static int Main(string[] args)
        {
                Console.WriteLine("1 C#\n  2  Managed C++\n  3  VB.NET");
                Console.WriteLine("please enter your implementation language");
                int n=Console.ReadLine();
                switch(n)
                {
                    case 1: Console.WriteLine("good choice C# is  all about managed code");
                            break;
                    case 2: Console.WriteLine("let me guess, maintaining a legacy system?");
                            break;
                    case 3: Console.WriteLine("VB.NET It is not just for kids anymore");
                            break;
                   default: Console.WriteLine("well good luck with that");
                            break;
                }
                return 0;
        }
}
```

COMPLETE SET OF OPERATORS

• Following is a set of operators provided by C#.

Operator Category	Operators
Unary	+, -, !, ~, ++, --
Multiplicative	*, /, %
Additive	+, -
Shift	<<, >>
Relational	<, >, <=. >=, is, as
Equality	==, !=
Logical	& (AND), ^ (XOR), | (OR)
Conditional	&&, ||, ?: (ternary operator)
Indirection/Address	*, ->, &
Assignment	=, *=, -=, +=, /=, %=, <<=, >>=, &=, ^=, |=

• "is" operator is used to verify at runtime if an object is compatible with a given type. One common use is: to determine if a given object supports a particular interface.
• "as" keyword is used to downcast between types or implemented interface.
• As C# supports inter-language interaction, it supports the C++ pointer manipulation operators like *, -> and &. But if we use any of these operators, we are going bypass the runtime memory management scheme and writing code in *unsafe mode.*

THE SYSTEM DATA TYPES (AND C# ALIASES)

• Every intrinsic data type is an alias to an existing type defined in the System namespace (Fig. 3.13).
• Specifically, each C# data type aliases a well-defined structure type in the System namespace.
• Following table lists each system data type, its range, the corresponding C# alias and the type's compliance with the CLS.

C# Alias	CLS Compliant?	System Type	Range	Meaning
sbyte	No	System.SByte	-128 to 127	Signed 8-bit number
byte	Yes	System.Byte	0 to 255	Unsigned 8-bit number
short	Yes	System.Int16	-2^{16} to $2^{16}-1$	Signed 16-bit number
ushort	No	System.UInt16	0 to $2^{32}-1$	Unsigned 16-bit number
int	Yes	System.Int32	-2^{32} to $2^{32}-1$	Signed 32-bit number
uint	No	System.UInt32	0 to $2^{64}-1$	Unsigned 32-bit number
long	Yes	System.Int64	-2^{64} to $2^{64}-1$	Signed 64-bit number
ulong	No	System.UInt64	0 to $2^{128}-1$	Unsigned 64-bit number
char	Yes	System.Char	U10000 to U1ffff	A Single 16-bit Unicode character
float	Yes	System.Single	1.5×10^{-45} to 3.4×10^{38}	32-bit floating point number
double	Yes	System.Double	5.0×10^{-324} to 1.7×10^{308}	64-bit floating point number
bool	Yes	System.Boolean	True or False	Represents truth or falsity
decimal	Yes	System.Decimal	1 to 10^{28}	96-bit signed number
string	Yes	System.String	Limited by system memory	Represents a set of Unicode characters
object	Yes	System.Object	Anything derive from object	The base class of all types in the .NET universe.

• From this table, we can see that all the types are ultimately derived from System.Object.
• Since the data types like *int* are simply shorthand notations for the corresponding system type (like System.Int32), the following statements are valid –

Console.WriteLine(25.GetHashCode());
Console.WriteLine(32.GetType().BaseType()); etc.

• We can see that, though C# defines a number of data types, only a subset of the whole set of data types are compliant with the rules of CLS.
• So, while building user-defined types (like class, structures), we should use only CLS-compliant types.
• And also, we should avoid using unsigned types as public member of user-defined type.
• By doing this, the user-defined type (class, enumeration, structure etc) can be understood by any language in .NET framework.

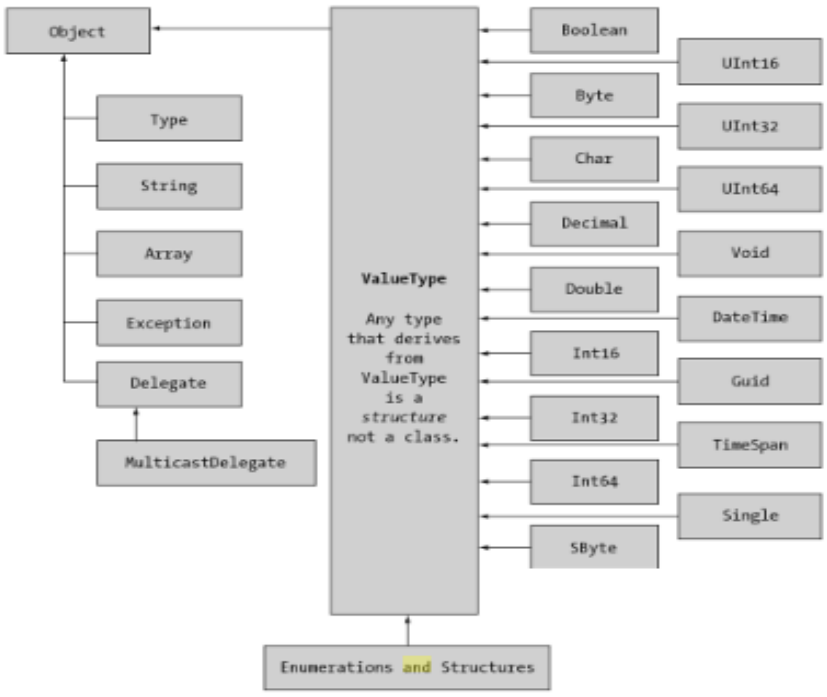


Figure 3-13. The hierarchy of System types

EXPERIMENTING WITH THE SYSTEM DATA TYPES

• The only purpose of System.ValueType is to override the virtual methods defined by System.Object to work with value-based versus reference-based semantics.

```
public abstract class ValueType: object
{
        public virtual bool Equals(object obj);
        public virtual int GetHashCode();
        public Type GetType();
        public virtual string ToString();
}
```

BASIC NUMERICAL MEMBERS

• The numerical types support MaxValue and MinValue properties that provide information regarding the minimum and maximum value a given type can hold.

• For example:

```
using System;
class Test
{
  public static void Main()
  {
   System.UInt16 a=30000;

   Console.WriteLine("Max value for UInt16: {0}", UInt16.MaxValue);          //65535
   Console.WriteLine("Min value for UInt16: {0}", UInt16.MinValue);          //0
   Console.WriteLine("value of UInt16: {0}", a);                             //30000
   Console.WriteLine("The type is: {0}", a.GetType().ToString());            //System.UInt16

   ushort b=12000;

   Console.WriteLine("Max value for ushort: {0}", ushort.MaxValue);          //65535
   Console.WriteLine("Min value for ushort: {0}", ushort.MinValue);          //0
   Console.WriteLine("value of ushort: {0}", b);                             //12000
   Console.WriteLine("The type is: {0}", b.GetType().ToString());            //System.UInt16

  }
}
```

MEMBERS OF SYSTEM.BOOLEAN

• The only valid assignment a bool can take is from the set {true | false}.

• We cannot assign make-shift values (e.g. 1 0 -1) to a bool.

• System.Boolean does not support a MinValue/MaxValue property-set but rather TrueString/FalseString.

MEMBERS OF SYSTEM.CHAR

• All the .NET-aware languages map textual data into the same underlying types viz System.String and System.Char, both are Unicode.

• The System.Char type provides several methods as shown in the following example:

```
using System;
class Test
{
        public static void Main()
        {
          bool b1=true;
          bool b2=false;

          Console.WriteLine("{0}", bool.FalseString);                        //False
          Console.WriteLine("{0}", bool.TrueString);                         //True
          Console.WriteLine("{0}, {1}", b1, b2);                             //True, False
          Console.WriteLine("{0}", char.IsDigit('P'));                       //False
          Console.WriteLine("{0}", char.IsDigit('9'));                       //True
          Console.WriteLine("{0}", char.IsLetter("10", 1));                  //False
          Console.WriteLine("{0}", char.IsLetter("1a", 1));                  //True
          Console.WriteLine("{0}", char.IsLetter('p'));                      //True
          Console.WriteLine("{0}", char.IsWhiteSpace("Hello World", 5));     //True
          Console.WriteLine("{0}", char.IsWhiteSpace("Hello World", 6));     //False
          Console.WriteLine("{0}", char.IsLetterOrDigit('?'));               //False
          Console.WriteLine("{0}", char.IsPunctuation('!'));                 //True
          Console.WriteLine("{0}", char.IsPunctuation('<'));                 //False
          Console.WriteLine("{0}", char.IsPunctuation(','));                 //True
        }
}
```

DEFINING PROGRAM CONSTANTS

• "const" keyword is used to define variables with a fixed, unaltered value.
• Unlike C++, "const" keyword cannot be used to qualify parameters or return values.
• The value of a constant point of data is computed at compile-time and therefore a constant-member cannot be assigned to an object-reference.
• It is possible to define local constants within a method scope.
• If we create a utility class that contains nothing but constant data, we may wish to define a private constructor. In this way, we ensure the object user cannot make an instance of the class.
• Private constructors prevent the creation of a given type.
• Consider the following code:

```
using System;
class MyConstants
{
        public const int myIntConst=5;
        public const string myStringConst="i am a constant";
        private MyConstants() { }
        public static void Main()
        {
                const string localConst="i am a rock,i am an island";
                Console.WriteLine("my integer constant={0}",myIntConst);
                Console.WriteLine("my string constant={0}",myStringConst);
                Console.WriteLine("local constant={0}",localConst);
        }
}
```

Output:

```
my integer constant=5
my string constant= i am a constant
local constant= i am a rock,i am an island
```

DEFINING CUSTOM CLASS METHODS

• A method exists to allow the type to perform a unit of work
• In C#, every data and a method must be a member of a class or structure. That is, we cannot have global data or method.
• The methods may or may not take parameters and they may or may not return a value.
• Also, custom methods (user defined methods) may be declared non-static (instance level) or static (class level).

METHOD ACCESS MODIFIERS

• Every method specifies its level of accessibility using following access modifiers:

Access Modifiers	Meaning
public	Method is accessible from an object or any subclass.
private	Method is accessible only by the class in which it is defined. *private* is a default modifier in C#.
protected	Method is accessible by the defining class and all its sub-classes.
internal	Method is publicly accessible by all types in an assembly, but not outside the assembly.
protected internal	Method's access is limited to the current assembly or types derived from the defining class in the current assembly.

• Methods that are declared public are directly accessible from an object instance via the dot operator.
• Private methods cannot be accessed by an object reference but instead are called internally by the object to help the instance get its work done (i.e. *private helper functions*).

UNDERSTANDING STATIC METHODS

• A method can be declared as *static*. When a method is static, it can be invoked directly from the class-level, without creating an object.
• This is the reason for making Main() function to be static.
• The another example is *WriteLine()* method. We will directly use the statement *Console.WriteLine()* without creating an object of *Console* class. For example:

```
using System;
class Test
{
        public static void disp()
        {
                Console.WriteLine("hello");
        }

        public static void Main()
        {
                Test.disp();        //calling method using class name itself
        }
}
```

DEFINING STATIC DATA

• Static data is shared among all object instances of the same type. Rather than each object holding a copy of a given field, a point of static data is allocated exactly once for all instances of the type.
• If one object changes the value, all types 'see' the change.
• Main use: to allow all objects to share a given value at the class level. For example:

```
using System;
class Test
{
        public static int p=0;

        public int incr()
        {
                return ++p;
        }

        public static void Main()
        {
                Test t1=new Test();
                Test t2=new Test();

                Console.WriteLine("p= {0}", t1.incr());          //1
                Console.WriteLine("p= {0}", t2.incr());          //2
                Console.WriteLine("p= {0}", t1.incr());          //3
        }
}
```

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."
- Consider the following program:

```
class ArrayTest
{
        static void Main()
        {
                // Declare the array of two elements:
                int[][] arr = new int[2][];

                // Initialize the elements:
                arr[0] = new int[5] { 1, 3, 5, 7, 9 };
                arr[1] = new int[4] { 2, 4, 6, 8 };

                // Display the array elements:
                for (int i = 0; i < arr.Length; i++)
                {
                        Console.Write("Element({0}): ", i);

                        for (int j = 0; j < arr[i].Length; j++)
                        {
                                Console.Write("{0} ", arr[i][j]);
                        }
                        Console.WriteLine();
                }
        }
}
```

Output:

```
Element(0): 1 3 5 7 9
Element(1): 2 4 6 8
```

SYSTEM.ARRAY BASE CLASS

- Every array in C# is derived from the class System.Array.

Member	Meaning
BinarySearch()	This method searches a (previously sorted) array for a given item. If the array is composed of user-defined data types, the type in question must implement the IComparer interface to engage in a binary search.
Clear()	This method sets a range of elements in the array to empty values (0 for value types; null for reference types).
CopyTo()	This method is used to copy elements from the source array into the destination array.
Length	This property is used to determine the number of elements in an array.
Rank	This property returns the number of dimensions of the current array.
Reverse()	This method reverses the contents of a one-dimensional array.
Sort()	This method sorts a one-dimensional array of intrinsic types. If the elements in the array implement the IComparer interface, we can also sort an array of user-defined data type .

- Consider the following example to illustrate some methods and/or properties of System.Array class

```
using System;
class Test
{
        public static void Main()
        {
                int[] arr=new int[5]{12, 0, 45, 32, 67};

                Console.WriteLine("Array elements are :");

                for(int i=0;i<arr.Length;i++)
                        Console.WriteLine("{0}\t", arr[i]);

                Array.Reverse(arr);

                Console.WriteLine("Reversed Array elements are :");
                for(int i=0;i<arr.Length;i++)
                        Console.WriteLine("{0}\t", arr[i]);

                Array.Clear(arr, 1, 3);

                Console.WriteLine("Cleared some elements :");
                for(int i=0;i<arr.Length;i++)
                        Console.WriteLine("{0}\t", arr[i]);
        }
}
```

Output:

```
Array elements are:
12      0       45      32      67
Reversed Array elements are:
67      32      45      0       12
Cleared some elements:
67      0       0       0       12
```

STRING MANIPULATION IN C#

- Data type *string* is an alias type for *System.String* class.
- This class provides a set of methods to work on strings. Following is a list of few such methods:

Member	Meaning
Length	This returns the length of the current string.
Contains()	This is used to determine if the current string object contains a specified string.
Concat()	This method returns a new string that is composed of two discrete strings.
CompareTo()	Compares two strings.
Copy()	Returns a fresh new copy of an existing string.
Format()	This is used to format a string literal using other primitives (i.e., numerical data and other strings)
Insert()	This method is used to receive a copy of the current string that contains newly inserted string data.
PadLeft() PadRight()	These return copies of the current string that has been padded with specific data.
Remove() Replace()	Use these methods to receive a copy of a string, with modifications (characters removed or replaced).
Substring()	This returns a string that represents a substring of the current string.
ToCharArray()	This returns a character array representing the current string.
ToUpper() ToLower()	These create a copy of a given string in uppercase or lowercase.

- Consider the following example:

```
using System;
class Test
{
        public static void Main()
        {
                System.String s1="This is a string";
                string s2="This is another string";

                if(s1==s2)
                    Console.WriteLine("Same strings");
                else
                   Console.WriteLine("Different strings");

                string s3=s1+s2;

                Console.WriteLine("s3={0}",s3);

                for(int i=0;i<s1.Length;i++)
                    Console.WriteLine("Char {0} is {1}\n",i, s1[i]);

                Console.WriteLine("Contains 'is'?: {0}", s1.Contains("is"));

                Console.WriteLine(s1.Replace('a',' '));
        }
}
```

Output:

```
Different strings
s3=This is a stringThis is another string
Char 0 is T     Char 1 is h     Char 2 is i     Char 3 is s
Char 4 is       Char 5 is I     Char 6 is s     Char 7 is
Char 8 is a     Char 9 is       Char 10 is s    Char 11 is t
Char 12 is r    Char 13 is I    Char 14 is n    Char 15 is g
Contains 'is'?: True
This is  string
```

ESCAPE CHARACTERS AND *"VERBATIM STRINGS"*

• Just like C, C++ and Java, C# also provides some set of escape characters as shown:

Character	Meaning
\'	Inserts a single quote into a string literal.
\"	Inserts a double quote into a string literal.
\\	Inserts a backslash into a string literal. This can be quite helpful when defining file paths.
\a	Triggers a system alert (beep).
\b	Triggers a backspace.
\f	Triggers a form feed.
\n	Inserts a new line (on Win32 platforms).
\r	Inserts a carriage return.
\t	Inserts a horizontal tab into the string literal
\v	Inserts a vertical tab into the string literal
\0	Represents NULL character.

• In addition to escape characters, C# provides the @-quoted string literal notation named as ***verbatim string*.** Using this, we can bypass the use of escape characters and define the literals.

• Consider the following program:

```
using System;
class Test
{
        public static void Main()
        {
                string s1="I said, \"Hi\"";
                Console.WriteLine("{0}",s1);
                s1="C:\\Notes\\DotNet\\Chapter3.doc";
                Console.WriteLine("{0}",s1);
                 string s2=@"C:\Notes\DotNet\Chapter3.doc";
                Console.WriteLine("{0}",s2);
        }
}
```

Output:

```
I said, "Hi"
C:\Notes\DotNet\Chapter3.doc
C:\Notes\DotNet\Chapter3.doc
```

USING SYSTEM.TEXT.STRINGBUILDER

• The value of a string cannot be modified once established. i.e. strings are immutable.

• The methods like *Replace()* may seems to change the content of the string, but actually, those methods just output a copy of the string and the original string remains the same. For example:

```
string s1="hello";
Console.WriteLine("s1={0}", s1);          //hello
string s2=s1.ToUpper();
Console.WriteLine("s2={0}", s2);          //HELLO
Console.WriteLine("s1={0}", s1);          //hello
```

• Thus, whenever we want to modify a string, we should have a new string to store the modified version. That is, every time we have to work on a copy of the string, but not the original.

• To avoid this in-efficiency, C# provides a class called *StringBuilder* contained in the namespace *System.Text*.

• Any modification on an instance of StringBuilder will affect the underlying buffer itself.

• Consider the following program:

```
using System;
using System.Text;
class Test
{
        public static void Main()
        {
                StringBuilder s1= new StringBuilder("hello");
                s1.Append(" world");
                Console.WriteLine("{0}",s1);
                string s2=s1.ToString().ToUpper();
                Console.WriteLine("{0}",s2);
        }
}
```

Output:

```
hello world
HELLO WORLD
```

ENUMERATIONS

• When number of values taken by a type is limited, it is better to go for symbolic names rather than numeric values.

• For example, the marital status of a person can be any one of *Married, Widowed, Unmarried, Divorced*. To have such symbolic names, C# provides enumerations –

```
enum M_Status
{
        Married,        //0
        Widowed,        //1
        Unmarried,      //2
        Divorced        //3
}
```

• In enumeration, the value for first symbolic name is automatically initialized to 0 and second to 1 etc. If we want to give any specific value, we can use –

```
enum M_Status
{
        Married =125,
        Widowed,        //126
        Unmarried,      //127
        Divorced        //128
}
```

or

```
enum M_Status
{
        Married =125,
        Widowed=0,
        Unmarried=23,
        Divorced=12
}
```

• By default, the storage type used for each item of enumeration is System.Int32. We can change it, if we wish –

```
enum M_Status: byte
{
        Married =125,
        Widowed=0,
        Unmarried=23,
        Divorced=12
}
```

• Enumerations can be used as shown below –

```
using System;
class Test
{
        enum M_Status: byte
        {
                Married =125,
                Widowed=0,
                Unmarried=23,
                Divorced=12
        }

        public static void Main()
        {
           M_Status p1, p2;

           p1=M_Status.Married;
           p2=M_Status.Divorced;

           if(p1==M_Status.Married)
                Console.WriteLine("p1 is married");                    // p1 is married

           if(p2==M_Status.Divorced)
                 Console.WriteLine("p2 is {0}", M_Status.Divorced);     //p2 is Divorced
        }
}
```

SYSTEM.ENUM BASE CLASS

• The C# enumerations are derived from System.Enum class.

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

• Consider the following example to illustrate some of the methods of Enum class.

```
using System;
class Test
{
    enum M_Status
    {
          Married ,
          Widowed,
          Unmarried,
          Divorced
    }

    public static void Main()
    {
          Console.WriteLine(Enum.GetUnderlyingType(typeof(M_Status)));
          Array obj =Enum.GetValues(typeof(M_Status));

          Console.WriteLine("This enum has {0} members", obj.Length);

          foreach(M_Status p in obj)
          {
             Console.WriteLine("String name: {0}", p.ToString());
             Console.WriteLine("int: ({0}),", Enum.Format(typeof(M_Status), p, "D"));
             Console.WriteLine("hex: ({0}),", Enum.Format(typeof(M_Status), p, "X"));
          }

          if(Enum.IsDefined(typeof(M_Status), "Widowed"))
                Console.WriteLine("Widowed is defined");

          M_Status p1 = (M_Status)Enum.Parse(typeof(M_Status), "Divorced");
          Console.WriteLine("p1 is {0}", p2.ToString());

          M_Status p2=M_Status.Married;
          if(p1<p2)
               Console.WriteLine("p1 has less value than p2");
          else
              Console.WriteLine("p1 has more value than p2");
    }
}
```

Output:

```
System.Int32
This enum has 4 members
String name: Married          int: (0)  hex: (00000000)
String name: Widowed          int: (1)  hex: (00000001)
String name: Unmarried        int: (2)  hex: (00000002)
String name: Divorced         int: (3)  hex: (00000003)
Widowed is defined
p1 is Divorced
p1 has more value than p2
```

DEFINING STRUCTURES

• Structures behave similar to class, except that structure-memory will be allocated in stack-area, whereas for class-memory will be allocated from heap-area.
• It can have member-data, member-methods, constructors (only parameterized) and they can implement interfaces.
• It is directly derived from *System.ValueType*.
• We can implement boxing and unboxing on structures just like as we do for any intrinsic data types.
• Consider the following program:

```
using System;
struct EMP
{
        public int age;
        public string name;

        public EMP(int a, string n)
        {
                age=a;
                name=n;
        }

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

class Test
{
        public static void Main()
        {
                EMP e=new EMP(25, "Raja");
                e.disp();

                object ob=e;            //boxing
                MyFun(ob);
        }

        public static void MyFun(object obj)
        {
                EMP t=(EMP)obj;         //unboxing

                Console.WriteLine("After boxing and un-boxing:");
                t.disp();
        }
}
```

Output:

```
Name =Raja,  Age =25
After boxing and un-boxing:
Name =Raja,  Age =25
```

DEFINING CUSTOM NAMESPACES

• We can define our own namespace i.e. user-defined namespace (or custom namespace).
• Whenever we want to group similar classes into a single entity, we can define a namespace.
• Assume we need to develop a program to show the features of several vehicles like car, bus and bike. Then, the classes for all these vehicles can be put under a namespace as shown in below code:

```
namespace Vehicle
{
        public class Car
        {
                //members of Car class
        }

        public class Bus
        {
                //members of Bus class
        }

        public class Bike
        {
                //members of Bike class
        }
}
```

• Now, the namespace Vehicle acts as a container for all these classes. If we want to create an object of any of these classes in any other application, we can simply write –

```
using System;
using Vehicle;     //note this
class Test
{
        public static void Main()
        {
                Car c=new Car();
                        -------
                        -------
        }
}
```

RESOLVING NAME CLASHES ACROSS NAMESPACES

• There may be situation where more than one namespace contains the class with same name. For example, we may have one more namespace like –

```
namespace MyVehicle
{
        public class Car
        {
                //members of Car class
        }
}
```

• When we include the namespaces MyVehicle and Vehicle, and try to create an object of Car class, we will get an error.
• To avoid this, we will use dot operator for combining namespace name & class name. For example:

```
using System;
using Vehicle;
using MyVehicle;
class Test
{
        public static void Main()
        {
                Car c=new Car();                        //Error!!! name conflict
                Vehicle.Car c1=new Vehicle.Car();
                MyVehicle.Car c2=new MyVehicle.Car();
                ---------
        }
}
```

DEFINING NAMESPACE ALIASES

• The ambiguity in the namespaces can also be resolved using alias names as shown –

```
using System;
using Vehicle;
using MyVehicle;
using MyCar=MyVehicle.Car;
class Test
 {
        public static void Main()
        {
                Car c1=new Car();
                MyCar c2=new MyCar();
                ---------
        }
 }
```

NESTED NAMESPACES

• We can nest one namespace within the other also. For example –

```
namespace Vehicle
{
        namespace MyVehicle
        {
                -----
                -----
        }
}

Or
        namespace Vehicle.MyVehicle
        {
                -------
        }
```

EXERCISES

1) What is System.Object? Explain the instance methods of Object Class. (6)
2) Explain the default behavior of System.Object with example. (6)
3) How do you override ToString()of System.Object? Explain with example. (6)
4) How do you override Equals()of System.Object? Explain with example. (6)
5) Explain 4 iteration constructs with example for each. (8)
6) Explain 2 control flow constructs with example. (6)
7) Draw a diagram to depict the hierarchy of System types and explain. (6)
8) List out & explain various method access modifiers. (4)
9) Explain the static method and static data with example for each. (6)
10) Explain 2 types of multi-dimensional arrays with example for each. (8)
11) List out & explain core members of System.Array class. (4)
12) List out & explain core members of System.String class. (4)
13) Explain the features of StringBuilder class with example. (6)
14) Explain escape characters and "verbatim strings" with example. (4)
15) List out & explain core members of System.Enum class. (4)
16) Explain (un)boxing custom structures with example. (4)
17) Explain the following with reference to namespace: (8)
    i) Defining custom namespaces
    ii) Resolving name clashes across namespaces
    iii) Defining namespace aliases

UNIT 5: EXCEPTION & OBJECT LIFETIME

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
```

THE TARGETSITE PROPERTY

• This property is used to determine various details about the method that threw a given exception.
• Printing the value of TargetSite will display the return type, name, and parameters of the method that threw the exception.
• This can also be used to obtain name of the class that defines the offending-method.

THE STACKTRACE PROPERTY

• This property is used to identify the sequence of calls that resulted in the exception.
• We never set the value of StackTrace, as it is set automatically at the time the exception is created.

THE HELPLINK PROPERTY

• This returns a URL to a help-file describing the error in detail.
• By default, the value managed by HelpLink property is an *empty string.*
We can fill this property with a relevant value (i.e. URL).
• Consider the following code to illustrate the properties TargetSite, StackTrace & HelpLink:

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

        public void Fun(int d)
        {
                try
                   {
                      Exception ex= new Exception("crossed limit!!!");
                      ex.HelpLink="g:\\Help.doc";
                      throw ex;
                   }
                catch(Exception e)
                   {
                     Console.WriteLine("Error caught");
                     Console.WriteLine("Class defining member= {0}", e.TargetSite.DeclaringType);
                     Console.WriteLine("Member type= {0}", e.TargetSite.MemberType);
                     Console.WriteLine("Member name={0}", e.TargetSite);
                     Console.WriteLine("Message={0}", e.Message);
                     Console.WriteLine("Stack={0}", e.StackTrace);
                     Console.WriteLine("Help link ={0}",e.HelpLink);
                   }

               //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:

```
Enter a number=127
Error caught!
Class defining member= Test
Member type= Method
Member name= Void Fun(Int32)
Message= crossed limit!!!
Stack= at Test.Fun(Int32)
Help link= g:\Help.doc
Speed is =127
```

CLR SYSTEM-LEVEL EXCEPTIONS (SYSTEM.SYSTEMEXCEPTION)

• The .NET base class library(BCL) defines many classes derived from System.Exception.
• For example, the System namespace defines core error-objects such as
→ DivideByZeroException
→ OutOfMemoryException
→ IndexOutOfRangeException
• Exceptions thrown by methods in the BCL are called *system-exceptions*.
• Main idea: When an exception-type derives from System.SystemException, it can be concluded that the exception was raised by .NET runtime (rather than by the custom code-base of the executing application).
• If you fail to handle a raised exception, the operating-system will trigger a *"last chance exception".*

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

CUSTOM APPLICATION-LEVEL EXCEPTIONS (SYSTEM.APPLICATIONEXCEPTION)

• All .NET exceptions are class types and
hence application-specific(or user defined) exceptions can be created.
• Main idea: To identify the source of the (nonfatal) error. When an exception-type derives from System.ApplicationException, it can be concluded that the exception was raised by code-base of the executing application rather than by .NET BCL.

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

• The relationship between exception-centric base classes are shown below (Figure 5.1) –

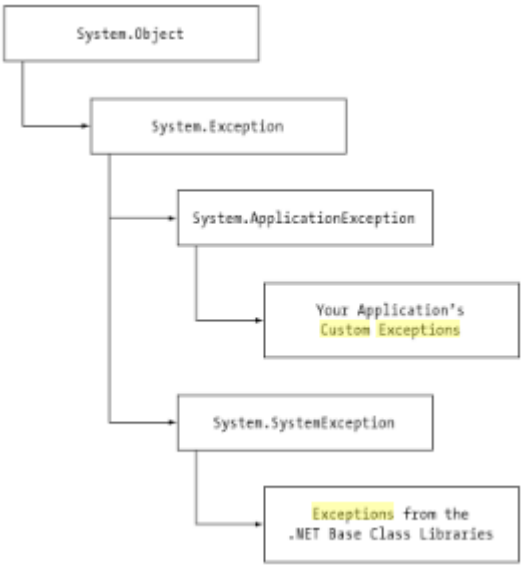


Figure 5.1: Application and system exceptions

BUILDING CUSTOM EXCEPTIONS, TAKE ONE

• We can always throw instances of System.Exception to indicate runtime-error.
But, it is better to build a custom class that encapsulates unique details of our current problem.

• Main use: We need to create custom exceptions only when the error is tightly bound to the class issuing the error.

• Like any class, the exception-class also may include fields, methods and properties that can be used within the catch-block.

• We can also override any virtual member in the parent class. For example, assume we wish to build a custom exception (named CarIsDeadException) to represent that the car has crossed the maximum speed limit.

```
public class CarIsDeadException : System.Exception
{
        private string messageDetails;
        public CarIsDeadException(){ }

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

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

public class Car
{
        ……….
        public void SpeedUp(int delta)
        {
                try
                {
                        speed=current_speed + delta;
                        if(speed>max_speed)
                                throw new CarIsDeadException("Ford Icon");
                }
                catch(CarIsDeadException e)
                {
                        Console.WriteLine("Method:{0}", e.TargetSite);
                        Console.WriteLine("Message:{0}", e. Message);
                }
        }
        ……..
}
```

BUILDING CUSTOM EXCEPTIONS, TAKE TWO

• We can write the constructors, methods and overridden properties as we wish in our exception class.

• But it is always recommended approach to build a relatively simple type that supplied three named constructors matching the following signature:

```
public class CarIsDeadException: System.Exception
{
    public CarIsDeadException() { . . .}
    public CarIsDeadException(string message):base(message) { . . .}
    public CarIsDeadException(string message,Exception innerEx):base(message,innerEx) { . . .}
}
```

• Most of the user-defined exceptions follow this pattern.

• Because, many times, the role of a custom exception is
→ not to provide additional functionality beyond what is provided by its base class. Rather,
→ to provide a strongly named type that clearly identifies the nature of the error.

BUILDING CUSTOM EXCEPTIONS, TAKE THREE

• The exceptions can be categorized as system-level or application-level types.
• If we want to clearly mark the exception to be thrown by the application itself and not by BCL, then we can redefine as –

```
public class CarIsDeadException: ApplicationException
{
    public CarIsDeadException() { . . .}
    public CarIsDeadException(string message):base(message) { . . .}
    public CarIsDeadException(string message,Exception innerEx):base(message,innerEx) { . . .}
}
```

HANDLING MULTIPLE EXCEPTIONS

• In reality, the code within try-block may trigger multiple possible exceptions.
• To catch multiple exceptions, we construct multiple catch-blocks for a single try-block
• When an exception is thrown, it will be processed by the "nearest available" catch.
• 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);
       }
        catch(OutOfMemoryException e)
       {
            Console.WriteLine("Error message = {0}", e.Message);
       }
        catch(Exception 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
```

• Here, the class Exception is a base class for all custom and system exceptions. Hence, it can handle any type of exceptions.
• If Exception is the first catch-block, the control will jump to that block itself and the other two exceptions are unreachable.
• Thus, we have to make sure the catch-blocks are structured such that
the very *first* catch is the *most specific* exception(or most derived type in inheritance hierarchy) whereas the final catch is the most general (or top most base class in inheritance hierarchy).

GENERIC CATCH STATEMENTS

• C# supports a generic catch block that does not explicitly define the type of exception. That is, we can write:

```
catch
{
        Console.WriteLine("Some error has occurred");
}
```

• But, using this type of catch blocks indicates that the programmer is un-aware of the type of exception that is going to occur, which is not acceptable. Hence it is always advised to use specific type of exceptions.

RETHROWING EXCEPTIONS

• To re-throw exception, simply make use of the "throw" keyword within a catch-block.

```
try
{ . . . }

catch(CarIsDeadException e)
{
    throw e
}
```

DYNAMICALLY IDENTIFYING APPLICATION AND SYSTEM LEVEL EXCEPTIONS

• It is possible to generalize the catch blocks in such a way that all application level exceptions are handled apart from possible system-level exceptions.

```
try
{
        //do something
}
catch(ApplicationException e)
{
   ---------------
}
catch(SystemException e)
{
    ---------------
}
```

• Though C# has the ability to discover at runtime the underlying source of an exception, we are gaining nothing by doing so.

• Because some BCL methods that should ideally throw a type derived from System.SystemException. are actually derived from System.ApplicationException or even more generic System.Exception.

EXERCISES

1. Differentiate between bugs, errors and exceptions. Explain the role of .NET exception handling. (6)
2. Define the following keywords with program: (6)
   i)try ii)throw iii)catch iv)finally
3. List out & explain core members of System.Exception class. (4)
4. With a program, explain & illustrate the use of System.Exception base class in throwing generic exceptions. (6)
5. With a program, explain & illustrate the use of System.Exception base class in catching exceptions. (6)
6. Briefly explain the usage of finally block. (4)
7. With a program, explain following properties: TargetSite, StackTrace, HelpLink. (6)
8. Compare System level exception vs. Application level exception. (4)
9. With a program, explain how to build custom exception. (6)
10. Write C# application to illustrate handling multiple exceptions. (4)
11. Why is proper ordering of catch blocks necessary in C#? (4)

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.

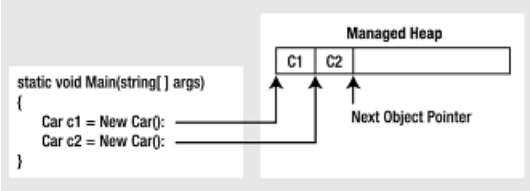


Figure 5.2: reference types are allocated on managed heap

THE BASICS OF GARBAGE COLLECTION

• Garbage collection is a process of reclaiming memory occupied by unreachable objects in the application.

• The golden rule of garbage collection:"If the managed-heap does not have sufficient memory to allocate a new object, garbage collection will occur".

• A root is a storage-location which contains a reference to an object on the heap.

The root can fall into following categories:

→ references to global/static objects

→ references to local objects within a given method.

• When a garbage-collection occurs, the CLR will inspect all objects on the heap to determine if it is still in use in the application.

• To do so, the CLR will build an object-graph.

Object-graph represents each object on the heap that is still reachable.

• When garbage-collector determines that a given root is no longer used by a given application, the object is marked for termination.

• When garbage-collector searches the entire heap for orphaned-root, the underlying memory is reclaimed for each unreachable object.

• Then, garbage-collector compacts empty block of memory on the heap (which in turn will cause CLR to modify the set of application roots to refer to the correct memory location).

• Finally, the NOP is re-adjusted to point to the next available slot.

• To illustrate this concept, assume that the heap contains a set of objects named A, B, C, D, E, F, and G (Figure 5.3).

• Let C and F are unreachable objects which are marked for termination as shown in the following diagram:

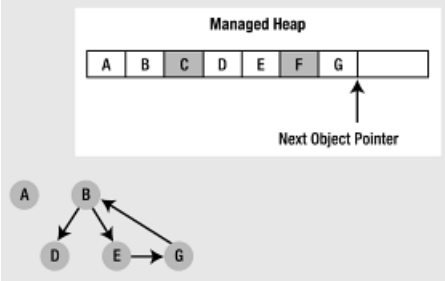


Figure 5.3: before garbage collection

• After garbage-collection, the NOP is readjusted to point to the next available slot as shown in the following diagram (Figure 5.4):

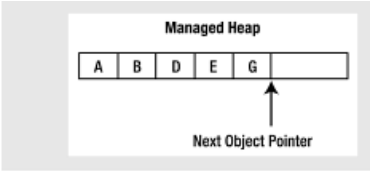


Figure 5.4: after garbage collection

FINALIZING A TYPE

• The .NET garbage collection scheme is *non-deterministic* in nature.
i.e. it is not possible to determine exactly when an object will be de-allocated from the memory.
• As a result, there is a possibility that the objects are holding *unmanaged resources* longer than necessary.
• When we build .NET types that interact with unmanaged resourceS, we like to ensure that these resources are released in-time rather than waiting for .NET garbage collector.
• To facilitate this, C# provides an option for overriding the virtual System.Object.Finalize() method.
• It is not possible to directly override the Finalize() using standard C# syntax as shown below:

```
public class Test
{
        protected override void Finalize()    //compile time error!!!
        {       …….             }
}
```

• Rather, a C++ type destructor syntax must be used as shown below:

```
public class Test
{
        ~Test()
        {
            Console.WriteLine("->finalizing test");
        }
}
```

• The C# destructor style syntax can be understood as a shorthand notation for the following code:

```
protected override void Test()
{
        try
        {
                Console.WriteLine("->finalizing test ")
                finally
                {base.Finalized(); }
        }
}
```

(INDIRECTLY) INVOKING SYSTEM.OBJECT.FINALIZE()

• Finalization will automatically take place when an application is unloaded by the CLR.
• Rule: When an application is unloaded, the Finalize() method is invoked for all finalizable objects.
• Consider the following example:

```
using System;
class Test
{
        public Test()
        { ……. }

        ~Test()
        {
                Console.WriteLine("Finalizing Test !!!");
        }

        public static void Main()
        {
                Console.WriteLine("Within Main()");
                Test t=new Test();
                Console.WriteLine("Exiting Main()");
        }
}
```

Output:

```
Within Main()
Exiting Main()
Finalizing Test!!!
```

THE FINALIZATION PROCESS

• Finalizer is used to ensure that an object can clean-up unmanaged resources.
• When an object is placed on a heap using *new*, the CLR automatically inspects whether this object supports a custom Finalize() method.
• If yes, the object is marked as 'finalizable' & a pointer to this object is stored in *Finalization-Queue.*
• Finalization-Queue is a table maintained by the CLR that points to every object that must be finalized before it is removed from the heap.
• When the garbage collector starts its action, it
→ inspects each entry in the FQ and
→ copies object off the heap to *finalization-reachable table* (f-reachable table).
• Then, a separate thread is generated to invoke the Finalize() for each object on f-reachable table at the next garbage collection.
• Drawback: This process will consume time and hence affects the performance of the application.

BUILDING AN ADHOC DESTRUCTION METHOD

• The process of finalizing an object is quite time consuming and hence affects the performance of the application.
• When a type manipulates unmanaged resources, it must be ensured that they are released in a timely and predictable manner.
• Solution: The programmer can write a custom adhoc method that can be invoked manually before the object goes out of scope.
• The custom method will take care of cleaning up the unmanaged resources.
• This method will
→ avoid the object being placed at finalization-queue and
→ avoid waiting for garbage collector to clean-up.

```
public class Car
{
        finally
        {
                public void Kill()          //name of method can be anything
                {
                   //clean up unmanaged resources
                }
        }
}
```

THE IDISPOSABLE INTERFACE

• In order to provide symmetry among all objects that support an explicit destruction method, the .NET BCL defines an interface named IDisposable.
• This interface supports a single member named Dispose().

```
public interface IDisposable
{
        public void Dispose();
}
```

• Now, the application can implement this interface and define Dispose() method.
• Rule: Always call Dispose() for any object you manually allocate to the heap. The assumption you should make is that if the class designer chose to support the Dispose() method, the type has some cleanup to perform.

```
public Car:IDisposable
{       public void Dispose()
        {
                //clean up your internal unmanaged resources
        }
}

public class App
{       public static int Main(string[] args)
        {
                Car c1=new Car("car one",40,10);
                c1.Dispose();
                return 0;
        }
}
```

• Dispose() method can be called manually before the object goes out of scope.
• This method will take care of cleaning up the unmanaged resources.
• This method will
→ avoid the object being placed at finalization-queue and
→ avoid waiting for garbage collector to clean-up

REUSING THE C# *USING* KEYWORD

• When we are using an object that implements IDisposable, it is quite common to use structured exceptions just to ensure the Dispose() method is called when exception occurs:

```
public void Test()
{
        Car c=new Car();
        try
        {       ……..    }
        catch
        {       ……..    }
        finally
        {       ……..
                c.Dispose();
        }
}
```

• C# provides another way of doing this with the help of *using* keyword:

```
public void Test()
{
        using(Car c=new Car())
        {
                //Do something
                //Dispose() method is called automatically when this block exits
        }
}
```

• One good thing here is, the *Dispose()* method is called automatically when the program control comes out of *using* block.

• But there is a disadvantage: If at all the object specified at *using* does not implement IDisposable, then we will get compile-time error.

GARBAGE COLLECTION OPTIMIZATION

• To locate unreachable objects, CLR does not inspect every object placed on heap to find orphaned-roots. Because, doing so will consume more time for larger applications.

• Each object on the heap is assigned to a given "generation".

• The main idea behind generation:

→The longer an object has existed on the heap; the more likely it is to stay there.
For example, application-level objects.

→Conversely, if an object is recently placed on heap; it may be dereferenced by application quickly.
For example, objects within a scope of a method.

• Each object belongs to one of following generations:

i) Generation-0(G0): Identifies a newly allocated object that has never been marked for garbage collection.
ii) Generation-1(G1): Identifies an object that has survived one garbage collection sweep.
iii) Generation-2(G2): Identifies an object that has survived more than one garbage collection sweep.

• Now, when garbage collection occurs, the GC marks and sweeps all generation-0 objects first.

• If this results in the required amount of memory, the remaining objects are promoted to the next available generation (G0->G1 & G1->G2).

• If all generation-0 objects have been removed from the heap, but more memory is still necessary, generation-1 objects are marked and swept, followed(if necessary) by generation-2 objects.

CORE MEMBERS OF SYSTEM.GC TYPE

• The programmer can interact with the garbage collector using a base class System.GC.

Member	Meaning
Collect ()	This forces the GC to call the Finalize() method for every object on managed-heap.
GetTotalMemory()	This returns the estimated amount of memory currently being used by all objects in the heap.
GetGeneration()	This returns the generation to which an object currently belongs.
MaxGeneration()	This returns the maximum generations supported on the target system.
ReRegisteredForFinalize()	This sets a flag indicating that the suppressed-object should be reregistered as finalize.
SuppersFinalize()	This sets a flag indicating that a given object's Finalize() method should not be called.
WaitForPendingFinalizers()	This suspends the current thread until all finalizable objects have been finalized. This method is typically called directly after invoking GC.Collect().

BUILDING FINALIZATION AND DISPOSABLE TYPES

• Consider an example to illustrate how to interact with .NET garbage collector.

```
using System;
class Car : IDisposable
{
        string name;

        public Car(string n)
        {
                name=n;
        }

        ~Car()
        {
                Console.WriteLine("Within destructor of {0}", name);
        }

        public void Dispose()
        {
                Console.WriteLine("Within Dispose() of {0}", name);
                GC.SuppressFinalize(this);
        }
}

class Test
{
    public static void Main()
    {
        Car c1=new Car("One");
        Car c2=new Car("Two");
        Car c3=new Car("Three");
        Car c4=new Car("Four");

        c1.Dispose();
        c3.Dispose();
     }
}
```

Output:

```
Within Dispose() of One
Within Dispose() of Three
Within destructor of Four
Within destructor of Two
```

• Both Dispose() and Finalize() (or destructor) methods are used to release the unmanaged resources.

• As we can see in the above example, when Dispose() method is invoked through an object, we can prevent the CLR from calling the corresponding destructor with the help of SuppressFinalize() method of GC class.

• By manually calling Dispose() method, we are releasing the resources and hence there is no need to call finalizer.

• Calling the Dispose() function manually is termed as *explicit object de-allocation* and making use of finalizer is known as *implicit object de-allocation.*

FORCING A GARBAGE COLLECTION

• We know that, CLR will automatically trigger a garbage collection when a managed heap is full.

• We, the programmers, will not be knowing, when this process will happen.

• However, if we wish, we can force the garbage collection to occur using the following statements:

GC.Collect();
GC.WaitForPendingFinalizers();

• The method *WaitForPendingFinalizers()* will allow all finalizable objects to perform any necessary cleanup before getting destroyed. Though, we can force garbage collection to occur, it is not a good programming practice.

PROGRAMMATICALLY INTERACTING WITH GENERATIONS

- We can investigate the generation of an object currently belongs to using GC.GetGeneration().
- GC.Collect() allows you to specify which generation should be checked for valid application roots.
- Consider the following code:

```
class Car:IDisposable
{
        string name;

        public Car(string n)
        {
                name=n;
        }

        ~Car()
        {
                Console.WriteLine("Within destructor of {0}", name);
        }

        public void Dispose()
        {
                Console.WriteLine("Within Dispose() of {0}", name);
                GC.SuppressFinalize(this);
        }
}

class Test
{
        public static void Main()
        {
                Car c1=new Car("One");
                Car c2=new Car("Two");
                Car c3=new Car("Three");
                Car c4=new Car("Four");

                Console.WriteLine("c1 is Gen {0}", GC.GetGeneration(c1));
                Console.WriteLine("c2 is Gen {0}", GC.GetGeneration(c2));
                Console.WriteLine("c3 is Gen {0}", GC.GetGeneration(c3));
                Console.WriteLine("c4 is Gen {0}", GC.GetGeneration(c4));

                c1.Dispose();
                c3.Dispose();

                GC.Collect(0);

                Console.WriteLine("c1 is Gen {0}", GC.GetGeneration(c1));
                Console.WriteLine("c2 is Gen {0}", GC.GetGeneration(c2));
                Console.WriteLine("c3 is Gen {0}", GC.GetGeneration(c3));
                Console.WriteLine("c4 is Gen {0}", GC.GetGeneration(c4));
        }
}
```

Output:

```
C1 is Gen 0
C2 is Gen 0
C3 is Gen 0
C4 is Gen 0
Within Dispose() of One
Within Dispose() of Three
C1 is Gen 1
C2 is Gen 1
C3 is Gen 1
C4 is Gen 1
Within Destructor of Four
Within Destructor of Two
```

EXERCISES

1. Explain object lifetime in .NET. (6)
2. Explain the CIL of *"new"*. (6)
3. Describe the role of .NET garbage collection. (6)
4. Explain finalizing a type in.NET. (4)
5. With program,explain how to invoke System.Object.Finalize() indirectly (4)
6. Explain finalization process in .NET. (6)
7. Explain adhoc destruction method. (4)
8. With a program, explain the use of IDisposable interface. (4)
9. Explain use of 'using' keyword with respect to garbage collection. (4)
10. Explain how garbage collection is optimized in .NET. (6)
11. List out & explain core members of System.GC class. (4)
12. With a program, explain how to build finalization & disposable types. (6)
13. With a program, explain how to interact with generations. (6)

UNIT 6: INTERFACES & COLLECTIONS

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!
}
```

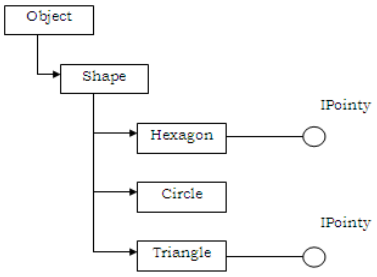
IMPLEMENTING AN INTERFACE

• 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 this new behavior:

```
// 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

UNDERSTANDING EXPLICIT INTERFACE IMPLEMENTATION

• Explicit interface implementation can be used to ensure that the methods defined by a given interface are only accessible from an interface-reference rather than an object-reference.
• Using this technique, we cannot make use of an access-modifier.
• Main purpose: to ensure that a given interface method is bound at the interface-level.
• Consider the following code:

```
public interface IDraw3D
{       void Draw();      }

public class Shape
{
        void Draw()
        {
                Console.WriteLine("drawing a shape");
        }
}

public class Line: Shape, IDraw3D
{
        void IDraw3D.Draw()
        {
                Console.WriteLine("drawing a 3D line");
        }
        public override void Draw()
        {
                Console.WriteLine("drawing a line");
        }
}

static void Main(string[] args)
{
        //This invokes the overridden Shape.Draw() method.
        Line myLine = new Line();
        myLine.Draw();

        // This triggers the IDraw3D.Draw() method.
        IDraw3D itfDraw3d= (IDraw3D)myLine;
        itfDraw3d.Draw();
}
```

• Interface as Polymorphic Agent: This can be very helpful when we want to implement a number of interfaces that happen to contain *identical methods.* For e.g.:

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

public interfaceIDraw3D
{       void Draw();       }

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

BUILDING INTERFACE HIERARCHIES

• Just as a class can serve as a base class to other classes, it is possible to build derived relationships among interfaces.

• The topmost interface defines a *general behaviour*
while the most derived interface defines more *specific behaviors*.

• Any methods defined by the base interfaces are automatically carried into the definition.

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

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

public interface IMetaFileRender : IPrintable
{
        void Render();
}

// This class supports IDrawable, IPrintable, and IMetaFileRender.
public class SuperImage : IMetaFileRender
{
        public void Draw()
        {
                Console.WriteLine("Basic drawing logic.");
        }
        public void Print()
        {
                Console.WriteLine("Draw to printer.");
        }
        public void Render()
        {
                Console.WriteLine("Render to metafile.");
        }
}

// Exercise the interfaces.
static void Main(string[] args)
{
        SuperImage si = new SuperImage();

        // Get IDrawable.
        IDrawable itfDraw = (IDrawable)si;
        itfDraw.Draw();

        // Now get ImetaFileRender, which exposes all methods up
        // the chain of inheritance.
        if (itfDraw is IMetaFileRender)
        {
                IMetaFileRender itfMF = (IMetaFileRender)itfDraw;
                itfMF.Render();
                itfMF.Print();
        }
}
```

Output:

```
Basic drawing logic
Render to metafile
Draw to printer
```

.

INTERFACES WITH MULTIPLE BASE INTERFACES

• An interface can derive from multiple base interfaces.
but a class cannot derive from multiple base classes.

• To illustrate, assume we are building a new set of interfaces that model the automobile behaviors:

```
interface IBasicCar
{       void Drive();           }

interface IUnderwaterCar
{        void Dive();           }

// Here we have an interface with TWO base interfaces.
interface IJamesBondCar: IBasicCar, IUnderwaterCar;
{        void TurboBoos();      }
```

• If we were to build a class that implements IJamesBondCar, we would now be responsible for implementing TurboBoost(), Dive(), and Drive():

```
public class JamesBondCar : IJamesBondCar
{
        public void Drive()
        {
                Console.WriteLine("Speeding up...");
        }
        public void Dive()
        {
                Console.WriteLine("Submerging...");
        }
        public void TurboBoost()
        {
                Console.WriteLine("Blast off!");
        }
}
```

• This specialized automobile can now be manipulated as we would expect:

```
static void Main(string[] args)
{
        JamesBondCar j = new JamesBondCar();
        j.Drive();
        j.TurboBoost();
        j.Dive();
}
```

Output:

```
Speeding up...
Blast off!
Submerging
```

.

UNDERSTANDING THE ICONVERTIBLE INTERFACE

• IConvertible type can be used to dynamically convert between data-types.

• Here is the formal definition:

```
public interface IConvertible
{
        bool ToBoolean(IFormatProvider provider);
        char ToChar(IFormatProvider provider);
        double ToDouble(IFormatProvider provider);
        string ToString(IFormatProvider provider);
        int ToInt32(IFormatProvider provider);
}
```

The IConvertible.ToXXXX() Members

• IConvertible interface defines a number of methods of the form ToXXXX().

• ToXXXX() methods provide a way to convert from one data-type into another.

• Sometimes, it may not always be possible to convert between data-types.

For example: from a Boolean into a DateTime.

• If conversion is semantically wrongly formed, the runtime will throw an InvalidCastException.

A Brief Word Regarding IFormatProvider

• ToXXXX() methods take a parameter of type *IFormatProvider.*

• Objects that implement this interface are able to format their contents based on culture-specific information.

• Here is the formal definition:

```
public interface IFormatProvider
{
        object GetFormat(Type formatType);
}
```

IConvertible.GetTypeCode()

• GetTypeCode() method can be used to programmatically discover a value that represents type-code of the data-type.

The System.Convert Type

• System namespace defines a data-type named *Convert*, which echoes the functionality of IConvertible interface.

• System.Convert does not directly implement IConvertible, however the same set of members are defined on its default public interface.

BUILDING A CUSTOM ENUMERATOR (IENUMERABLE & IENUMERATOR)

• IEnumerable interface is found within the System.Collections namespace.

• GetEnumerator() method returns a reference to yet another interface named System.Collections.IEnumerator.

• IEnumerable interface allows the caller to traverse the internal objects contained by the IEnumerable-compatible container.

• Here is the formal definition:

```
//This interface informs the caller that the object's subitems can be enumerated.
public interface IEnumerable
{
        IEnumerator GetEnumerator();
}
```

```
// This interface allows the caller to obtain a container's subitems.
public interface IEnumerator
{
    bool MoveNext ();          // Advance the internal position of the cursor.
    object Current { get;}     // Get the current item (read-only property).
    void Reset ();             // Reset the cursor before the first member.
}
```

• The System.Array type already implements IEnumerable & IEnumerator. Consider the following code:

```
using System.Collections;
public class Garage : IEnumerable
{
        // System.Array already implements IEnumerator!
        private Car[] carArray;

        public Garage()
        {
                carArray = new Car[4];
                carArray[0] = new Car("FeeFee", 200, 0);
                carArray[1] = new Car("Clunker", 90, 0);
                carArray[2] = new Car("Zippy", 30, 0);
                carArray[3] = new Car("Fred", 30, 0);
        }

        public IEnumerator GetEnumerator()
        {
                // Return the array object's IEnumerator.
                return carArray.GetEnumerator();
        }
}
```

• Once we have updated our Garage type, we can now safely use the type within the foreach loop.

• Furthermore, given that the GetEnumerator() method has been defined publicly, the object user could also interact with the IEnumerator type:

```
static void Main(string[] args)
{
    // Manually work with IEnumerator.
    IEnumerator i = carLot.GetEnumerator();
    i.MoveNext();
    Car myCar = (Car)i.Current;
    Console.WriteLine("{0} is going {1} MPH", myCar.PetName,myCar.CurrSpeed);
}
```

BUILDING CLONEABLE OBJECTS (ICLONEABLE)

• System.Object defines a member named MemberwiseClone(). This member is used to make a shallow copy of an object instance.
• Object-users do not call this method directly however, a given object instance may call this method itself during the cloning process.
• Setting one reference to another reference results in 2 variables pointing to the same object in memory.
• When we wish to equip our custom types to support the ability to return an identical copy of itself to the caller, we may implement the standard ICloneable interface.
• Here is the formal definition:

```
public interface ICloneable
{
        object Clone();
}
```

```
// The Point now supports "clone-ability."
public class Point : ICloneable
{
        public int x, y;
        public Point(){ }
        public Point(int x, int y)
        {
                this.x = x; this.y = y;
        }

        // Return a copy of the current object.
        public object Clone()
        {
                return new Point(this.x, this.y);
        }

        public override string ToString()
        {
                return string.Format("X = {0}; Y = {1}", x, y );
        }
}

static void Main(string[] args)
{
        // Notice Clone() returns a generic object type.
        // You must perform explicit cast to obtain the derived type.
        Point p3 = new Point(100, 100);
        Point p4 = (Point)p3.Clone();
        // Change p4.x (which will not change p3.x).
        p4.x = 0;

        // Print each object.
        Console.WriteLine(p3);
        Console.WriteLine(p4);
}
```

Output:

```
X = 100; Y = 100
X = 0; Y = 100
```

BUILDING COMPARABLE OBJECTS (ICOMPARABLE)

• This interface can be used to sort an object based on some internal-key.

• When we invoke Sort() method on an array of intrinsic types, we are able to sort the items in the array from lowest to highest.

• The logic behind CompareTo() is to test the incoming type against the current instance.

Table 6.1 : CompareTo() return values

Zero	If this instance is equal to object
Any number less than zero	If this instance is less than object
Any number greater than zero	If this instance is greater than object

```
interface IComparable       // the formal definition
{
        int CompareTo(object o);
}

public class Car
{
        private int carID;
        public int ID
        {
                get { return carID; }
                set { carID = value; }
        }
        public Car(string name, int currSp, int id)
        {
                currSpeed = currSp;
                petName = name;
                carID = id;
        }
}
public class Car: IComparable
{
        int IComparable.CompareTo(object o)
        {
                Car temp=(Car)o;
                if(this.CarID>temp.CarID)
                        return 1;
                if(this.CarID<temp.CarID)
                        return -1;
                else
                        return 0;
        }
}
public class CarApp
{
        public static int Main(string[] args)
        {
                Car[] myAutos=new Car[3];
                myAutos[0]=new Car(112,"mary");
                myAutos[1]=new Car(11,"jimmy");
                myAutos[2]=new Car(21,"rusty");
                Consoe.WriteLine("here is the unordered set of cars");
                foreach(Car c in myAutos)
                Console.WriteLine("{0} {1}",c.ID,c.PetName);

                Array.Sort(myAutos);

                Console.WriteLine("here is the ordered set of cars");
                foreach(Car c in myAutos)
                Console.WriteLine("{0} {1}",c.ID, c.PetName);
                return 0;
        }
}
```

Output:

```
here is the unordered set of cars
112 mary
11 jimmy
```

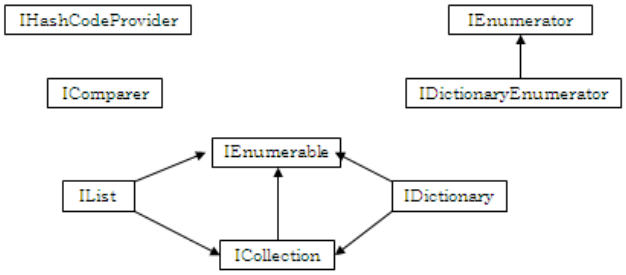
```
21 rusty
here is the ordered set of cars
11 jimmy
21 rusty
112 mary
```

EXPLORING THE SYSTEM.COLLECTIONS NAMESPACE

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

System.Collections Interface	Meaning
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.
IDictionaryEnumerator	Enumerates the contents of a type supporting IDictionary.
IEnumerable	Returns the IEnumerator interface for a given object.
IEnumerator	Generally supports foreach-style iteration of subtypes.
IHashCodeProvider	Returns the hash code for the implementing type using a customized hash algorithm.
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:



THE ROLE OF ICOLLECTION

• This interface provides a small set of properties which ca n be used to determine:
  a) Number of items in the container
  b) The thread-safety of the container
  c) The ability to copy the contents into a System.Array type.

• Here is the formal definition:

```
public interface ICollection: IEnumerable
{
        int Count {get;}
        bol IsSynchronized { get;}
        object SyncRoot{ get; }
        void CopyTo(Array array, int index);
}
```

THE ROLE OF IDICTIONARY

• A dictionary is a collection that allows an object to represent its contents using name/value pairs.

• Here is the formal definition:

```
public interface IDictionary: ICollection, IEnumerable
{
        bool IsFixedSize { get; }
        ICollection Keys { get;}
        ICollection Values { get;}
        void Add(object key, object value);
        void Remove(object key);
        void Clear();
}
```

THE ROLE OF IDICTIONARYENUMERATOR

• This interface allows to list out items in the dictionary via the generic Entry property, which returns a System.Collections.DictionaryEntry type.
• In addition, we are also able to traverse the name/value pairs using the key/value properties.
• Here is the formal definition:

```
public interface IDictionaryEnumerator: IEnumerator
{
        DictionaryEntry Entry { get;}
        object Key { get;}
        object Value { get;}
}
```

THE ROLE OF IHASHCODEPROVIDER

• This interface provides the ability to retrieve the hash code for a particular type.
• Here is the formal definition:

```
public interface IHahCodeProvider
{
        int GetHashCode(object obj);
}
```

THE ROLE OF ILIST

• This provides the ability to insert, remove and index items into (or out of) a container.
• Here is the formal definition:

```
public interface IList: ICollection, IEnumerable
{
        bool IsFixed{ get;}
        int Add(object value);
        void Remove(object value);
        void Clear();
        void Insert(int index, object value)
}
```

THE CLASS TYPES OF SYSTEM.COLLECTIONS

System.Collections Class	Meaning	Key Implemented Interfaces
ArrayList	Represents a dynamically sized array of objects.	IList, ICollection, IEnumerable, and ICloneable
Hashtable	Represents a collection of objects identified by a numerical key. Custom types stored in a Hashtable should always override System.Object.GetHashCode().	IDictionary, ICollection, IEnumerable, and ICloneable
Queue	Represents a standard first-in, first-out (FIFO) queue.	ICollection, ICloneable, and IEnumerable
SortedList	Like a dictionary; however, the elements can also be accessed by ordinal position (e.g., index).	IDictionary, ICollection, IEnumerable, and ICloneable
Stack	A last-in, first-out (LIFO) queue providing push and pop (and peek) functionality.	ICollection, ICloneable, and IEnumerable

WORKING WITH THE ARRAYLIST TYPE

• ArrayList represents a dynamically sized array of objects.

• Insert() allows to plug a new item into the ArrayList at a specified index.

```
static void Main(string[] args)
{
        // Create ArrayList and fill with some initial values.
        ArrayList carArList = new ArrayList();
        carArList.AddRange(new Car[ ] { new Car("Fred", 90, 10), new Car("Mary",100, 50), new
        Car("MB", 190, 11)});

        Console.WriteLine("Items in carArList: {0}", carArList.Count);
        // Print out current values.
        foreach(Car c in carArList)
                Console.WriteLine("Car pet name: {0}", c.PetName);

        // Insert a new item.
        Console.WriteLine("\n->Inserting new Car.");
        carArList.Insert(2, new Car("TheNewCar", 0, 12));
        Console.WriteLine("Items in carArList: {0}", carArList.Count);

        // Get object array from ArrayList and print again.
        object[] arrayOfCars = carArList.ToArray();

        for(int i = 0; i < arrayOfCars.Length; i++)
        {
                Console.WriteLine("Car pet name:{0}",((Car)arrayOfCars[i]).PetName);
        }
}
```

• Here we are making use of the AddRange() method to populate our ArrayList with a set of Car types (this is basically a shorthand notation for calling Add() n number of times).

• Once we print out the number of items in the collection (as well as enumerate over each item to obtain the pet name), we invoke Insert().

• The Insert() allows to plug a new item into the ArrayList at a specified index.

• Finally, notice the call to the ToArray() method, which returns a generic array of System.Object types based on the contents of the original ArrayList.

WORKING WITH THE QUEUE TYPE

• Queues are containers that ensures that items are accessed using a first-in, first-out manner.

Member of Queue type	Meaning
Dequeue()	Removes and returns the object at the beginning of the Queue
Enqueue()	Adds an object to the end of the Queue
Peek()	Returns the object at the beginning of the Queue without removing it

```
public static void WashCar(Car c)
{
        Console.WriteLine("cleaning {0}",c.PetName);
}

static void Main(string[] args)
{
    // make a Q with three items
    Queue carWashQ=new Queue();
    carWashQ.Enqueue(new Car("firstcar",0,0));
    carWashQ.Enqueue(new Car("secondcar",0,0));
    carWashQ.Enqueue(new Car("thirdcar",0,0));
    Console.WriteLine("first in Q is{0}",((Car)carWashQ.Peek()).PetName);
    //remove each item from Q
    WashCar((Car)carWashQ.Dequeue());
    WashCar((Car)carWashQ.Dequeue());
    WashCar((Car)carWashQ.Dequeue());
    //try to de-Q again?
    try
    {
        WashCar((Car)carWashQ.Dequeue());
    }
    catch(Exception e)
    {
        Console.WriteLine("error {0}",e.Message);
    }
}
```

Output

```
first in Q is firstcar
cleaning firstcar
cleaning secondcar
cleaning thirdcar
error! Queue empty
```

WORKING WITH THE Stack TYPE

- Stacks represent a collection that maintains items using a last-in, first-out manner.
- It defines following members:

Member of Stack type	Meaning
Push()	Used to place items onto the stack
Pop()	Used to remove items from the stack.
Peek()	Returns the object at the beginning of the stack without removing it.

```
static void Main(string[] args)
{
        ...
        Stack stringStack = new Stack();
        stringStack.Push("One");
        stringStack.Push("Two");
        stringStack.Push("Three");
        // Now look at the top item, pop it, and look again.
        Console.WriteLine("Top item is: {0}", stringStack.Peek());
        Console.WriteLine("Popped off {0}", stringStack.Pop());
        Console.WriteLine("Top item is: {0}", stringStack.Peek());
        Console.WriteLine("Popped off {0}", stringStack.Pop());
        Console.WriteLine("Top item is: {0}", stringStack.Peek());
        Console.WriteLine("Popped off {0}", stringStack.Pop());
        try
        {
                Console.WriteLine("Top item is: {0}", stringStack.Peek());
                Console.WriteLine("Popped off {0}", stringStack.Pop());
        }
        catch(Exception e)
        {  Console.WriteLine("Error!! {0}", e.Message); }
}
```

Output:

```
top item is:three
popped off:three
top item is:two
popped off:two
top item is:one
popped off:one
error! Stack empty
```

SYSTEM.COLLECTIONS.SPECIALIZED NAMESPACE

System.Collections.Specialized	Meaning
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
NameObjectCollectionBase KeysCollection	Represents a collection of the string keys of a 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

EXERCISES

1) Bring out the differences between interface and abstract base classes. (6)
2) How do you implement an interface? Explain with an example. (6)
3) Explain different techniques for invoking interface members at object level. (6)
4) Explain with example, explicit interface implementation. (6)
5) Explain with example, interface hierarchy. (6)
6) Explain with example, multiple base interfaces. (4)
7) How do you pass interface as parameter to a method? Explain with an example. (4)
8) Explain the concept of interface-based polymorphism. (4)
9) Explain any two methods of IConvertible interface. (6)
10) Explain usage of IEnumerable & IEnumerator interfaces with suitable examples. (6)
11) What do you mean by cloneable object? Write an example to depict the implementation of ICloneable Interface. (6)
12) Illustrate with an example, implementation of IComparable interface. (6)
13) List out & explain interfaces provided by System.Collection namespace. Draw diagram to show hierarchical relationship that exists b/w these interfaces. (6)
14) What are the major classes in System.Collections namespace? Explain. (4)
15) Write a code segment to illustrate the working of ArrayList class. (4)
16) What the members of System.Collection.Queue? Explain. (4)
17) Write a code snippet to show the usage of Stack class. (4)