This lecture notes is prepared according to the syllabus of the following subjects –

- **C# Programming and .NET (10IS761), a 7th semester BE Information Science and Engineering syllabus according to 2010 scheme**
- **C# Programming and .NET (10CS761), a 7th semester BE Computer Science and Engineering syllabus according to 2010 scheme**
- **Topics in Enterprise Architecture – II (10MCA53 and 07MCA53), a 5th semester MCA syllabus according to 2010 and 2007 schemes**

All these syllabi have same contents. This document contains up to 5 units of the syllabus. It was initially prepared in the year 2009 and then updated frequently.

Few of the books referred for preparing this notes include:

- **Pro C# with .NET 3.0, 4th Edition, by Andrew Troelsen**
- **C# The Complete Reference, by Herbert Shildt**

UNIT 1. The Philosophy of .NET

However good a programming language is, it will be overshadowed by something better in the course of time. The languages like C, C++, Visual Basic, Java, the frameworks like MFC (Microsoft Foundation Classes), ATL(Active Template Library), STL(Standard Template Library) and the architectures like COM (Component Object Model), CORBA (Common Object Request Broker Architecture), EJB (Enterprise JavaBeans) etc. were possessing technologies which is needed by the programmer. But still, the hunt for better one will be going on even in future too.

1.1 Understanding the Previous State of Affairs

Before examining the specifics of the .NET universe, it's helpful to consider some of the issues that motivated the genesis of Microsoft's current platform. To get in the proper idea, let us discuss some of the limitations of previous technologies.

Life as a C/Win32 API Programmer

- Building windows applications using the raw API (Application Programming Interface) and C is a complex task.
- C is a very abrupt language.
- C developers have to perform manual memory management.
- C involves ugly pointer arithmetic, and ugly syntactical constructs.
- C is a structured language and so lacks the benefits provided by the object-oriented approach.
- When you combine the thousands of global functions and data types defined by the Win32 API to a difficult language like C, bugs will increase rapidly.

Life as a C++/MFC Programmer

- The improvement over raw C/API development is the use of the C++ programming language.
- Though C++ provides OOPs concepts like encapsulation, inheritance and polymorphism, it is not away from manual memory management and pointers.
- Even with difficulty, many C++ frameworks exist today. For example, the Microsoft Foundation Classes (MFC) provides the developer with a set of C++ classes that facilitate the construction of Win32 applications.
- The main role of MFC is to wrap a "sane subset" of the raw Win32 API behind a number of classes, magic macros, and numerous code-generation tools.
- Regardless of the helpful assistance offered by the MFC framework, C++ programming remains a difficult and 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, and simpler data access logic.

- Even more than MFC, VB6 hides the complexities of the raw Win32 API from view using a number of integrated code wizards, intrinsic data types, classes, and VB-specific functions.
- The major downfall of VB6 is that it is not a fully object-oriented language. For example, VB6 does not allow the programmer to establish "is-a" relationships between types.
- VB6 has no intrinsic support for parameterized class construction.
- VB6 doesn't provide the ability to build multithreaded applications.

Life as a Java/J2EE Programmer

- Java has greater strength because of its platform independence nature.
- Java cleans up many unsavory syntactical aspects of C++.
- Java provides programmers with a large number of predefined "packages" that contain various type definitions.
- Although Java is a very elegant language, one potential problem is that using Java typically means that you must use Java front-to-back during the development cycle.
- So, language integration is difficult in Java, which is against its primary goal (a single programming language for every need).
- Pure Java is simply not appropriate for many graphically or numerically intensive applications.
- For graphics oriented product, Java works slowly and compared to this, C++ or C would execute faster.
- Java provides a limited ability to access non-Java APIs and hence, there is little support for true cross-language integration.

Life as a COM Programmer

- COM is a group of classes, which works as a block of reusable code.
- The binary COM server can be accessed in a language-independent manner. For example, COM classes written using C++ can be used by VB6.
- But, COM's language independence is somewhat limited as it will not support inheritance. Rather, one must make use of "has-a" relationship to reuse COM class types.
- Although COM can be considered a very successful object model, it is extremely complex under the internally.
- To help simplify the development of COM binaries, numerous COM-aware frameworks have come into existence.
- Even if we choose a relatively simple COM-aware language such as VB6, we are still forced to contend with fragile registration entries and numerous deployment-related issues.

Life as a Windows DNA Programmer

- The popularity of Web applications is ever expanding.
- Sadly, building a web application using COM-based Windows DNA (Distributed interNet Applications) is quite complex.

- Some of this complexity is because Windows DNA requires the use of numerous technologies and languages like ASP, HTML, XML, JavaScript, VBScript, ADO etc.
- Many of these technologies are completely unrelated from a syntactic point of view.
- Also each language and/or technology has its own type system. For example, an “int” in JavaScript is not quite the same as an “Integer” in VB6.

1.2 The .NET Solution

For various problems faced in previous technologies, the .NET provides the solution. The .NET framework is a completely new model for building systems on the Windows family of operating systems, as well as on numerous non-Microsoft operating systems such as Mac OS X and various Unix/Linux distributions. Some core features provided by .NET are as follows:

- ***Full interoperability with existing code*:** Existing COM binaries can mingle/interop with newer .NET binaries and vice versa. Also, Platform Invocation Services allows to call C-based libraries (including the underlying API of the operating system) from .NET code.
- ***Complete and total language integration*:** Unlike COM, .NET supports cross-language inheritance, cross-language exception handling, and cross-language debugging.
- ***A common runtime engine shared by all .NET-aware languages*:** One aspect of this engine is a well-defined set of types that each .NET-aware language “understands.”
- ***A base class library*:** This library protects the programmer from the complexities of raw API calls and offers a consistent object model used by all .NET-aware languages.
- ***No more COM plumbing*:** IClassFactory, IUnknown, IDispatch, IDL code, and VARIANT-compliant data types (BSTR, SAFEARRAY, and so forth) have no place in a native .NET binary.
- ***A truly simplified deployment model*:** Under .NET, there is no need to register a binary unit into the system registry. Furthermore, .NET allows multiple versions of the same *.dll to exist on a single machine.

NOTE: The .NET platform has nothing to do with COM. In fact, the only way .NET and COM types can interact with each other is using the interoperability layer.

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

The entities that make .NET to provide several benefits are CLR, CTS, and CLS. The .NET can be understood as a new runtime environment and a comprehensive base class library.

CLR:

- The runtime layer is properly referred to as the ***Common Language Runtime* (*CLR*)**.
- The primary role of the CLR is to locate, load, and manage .NET types.
- The CLR also takes care of a number of low-level details such as memory management and performing security checks.

CTS:

- The ***Common Type System (CTS)*** specification fully describes the entities like all possible data types and programming constructs supported by the runtime.
- It specifies how these entities can interact with each other.
- It also specifies how they are represented in the .NET metadata format.

CLS:

- A given .NET –aware language might not support each and every feature defined by the CTS.
- The ***Common Language Specification*** **(*CLS*)** is a related specification that defines a subset of common types and programming constructs that all .NET programming languages can agree on.
- Thus, 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.

1.4 The Role of the Base Class Libraries

- In addition to the CLR and CTS/CLS specifications, the .NET platform provides a base class library that is available to all .NET programming languages.
- This base class library encapsulates various primitives such as threads, file input/output (I/O), graphical rendering, and interaction with various external hardware devices.
- It also provides support for a number of services required by most real-world applications.
- For example, the base class libraries define types that facilitate database access, XML manipulation, programmatic security, and the construction of web-enabled, traditional desktop and console-based front ends.
- From a high level, you can visualize the relationship between the CLR, CTS, CLS, and the base class library, as shown in Figure 1.1.

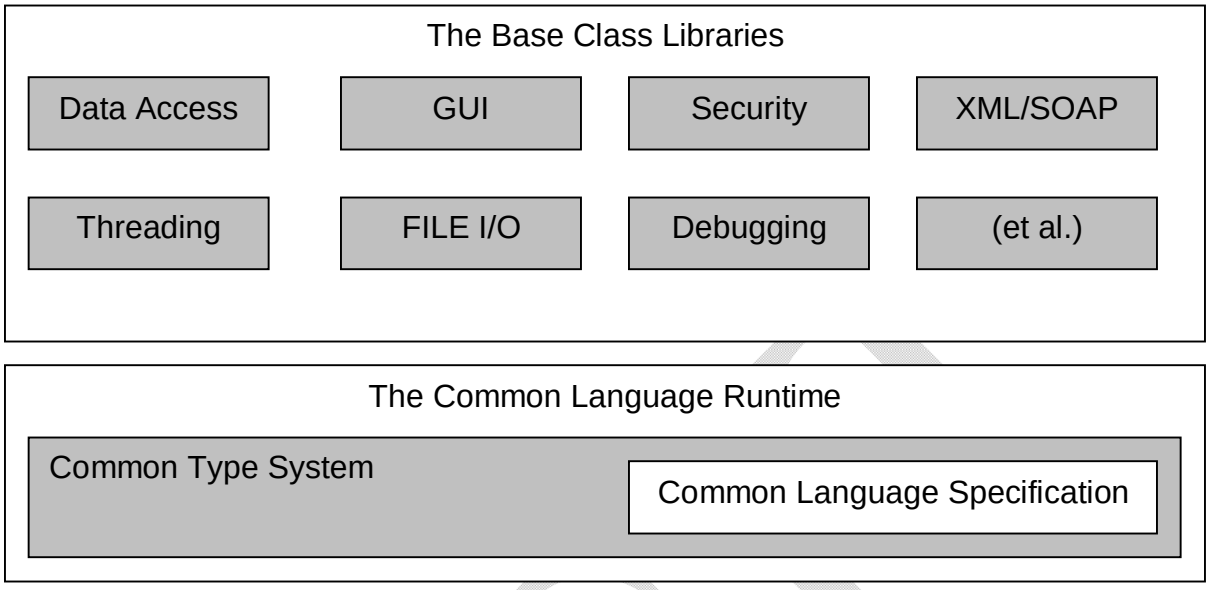


Figure 1.1 *The CLR, CTS, CLS, and base class library relationship*

1.5 What C# Brings to the Table

C# is a programming language that looks *very* similar to the syntax of Java. Many of the syntactic constructs of C# are taken from various aspects of Visual Basic 6.0 and C++. For example, like VB6, C# supports the notion of formal type properties, and the ability to declare methods taking varying number of arguments. Like C++, C# allows you to overload operators, as well as to create structures, enumerations, and callback functions (via delegates).

The features of C# language can be put together as –

- No pointers 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.
- The C++-like ability to overload operators for a custom type, without the complexity.
- The syntax for building generic types and generic members is very similar to C++ templates.
- Full support for interface-based programming techniques.
- Full support for aspect-oriented programming (AOP) techniques via attributes. This brand of development allows you to assign characteristics to types and their members to further qualify their behavior.

C# produces the code that can execute within the .NET runtime. The code targeting the .NET runtime is called as ***managed code***. The binary unit that contains the managed code is termed as ***assembly***. Conversely, code that cannot be directly hosted by the .NET runtime is termed ***unmanaged code*.**

1.6 An Overview of .NET Assemblies

- The .NET binaries are not described using COM type libraries and are not registered into the system registry.
- The .NET binaries do not contain platform-specific instructions, but rather platform-agnostic *intermediate language* (*IL*) and type metadata.

  (Note The terms IL, MSIL (Microsoft Intermediate Language) and CIL (Common Intermediate Language) are all describing the same exact entity.)

  The relationship between .NET aware compilers and metadata are shown in Fig. 1.2.

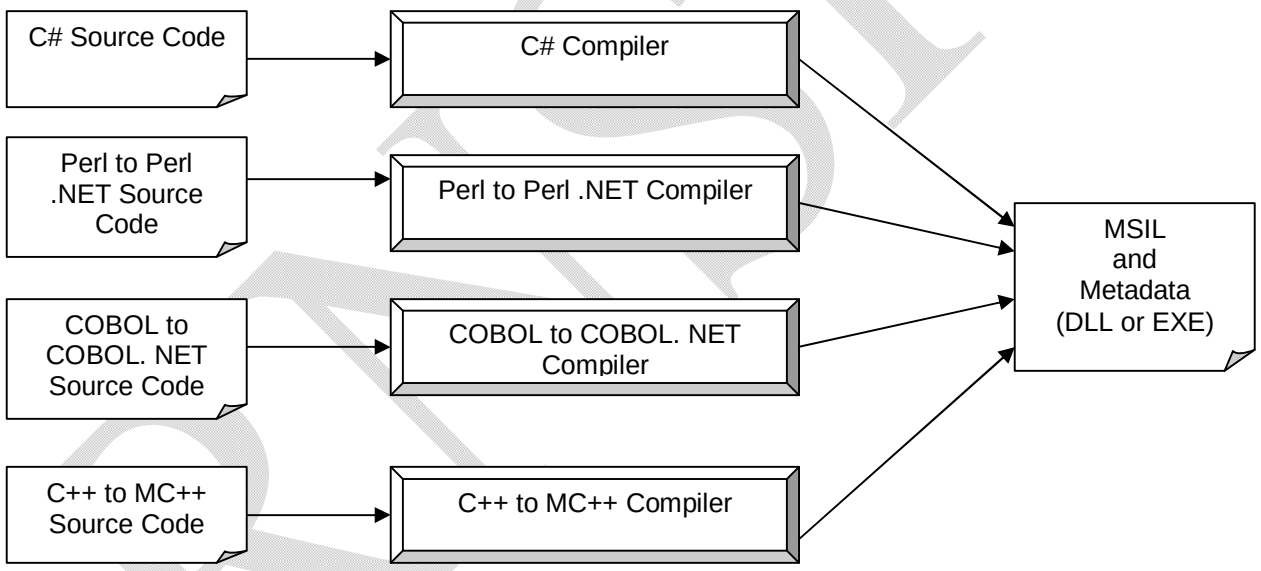


Figure 1.2 *All .NET-aware compilers produce IL instructions and metadata.*

- When a *.dll or *.exe has been created using a .NET-aware compiler, the resulting module is bundled into an ***assembly*.**
- An assembly contains CIL code, (which is conceptually similar to Java bytecode). This is not compiled to platform-specific instructions until absolutely necessary.
- When a block of CIL instructions (such as a method implementation) is referenced for use by the .NET runtime engine, CIL code will be compiled.
- Assemblies also contain *metadata* that describes the characteristics of every "type" living within the binary, in detail. For example, if you have a class named SportsCar, the type

metadata describes details such as SportsCar's base class, which interfaces are implemented by SportsCar (if any), as well as a full description of each member supported by the SportsCar type.

- Unlike COM, the .NET metadata is always present and is automatically generated by a given .NET-aware compiler.
- In addition to CIL and type metadata, assemblies themselves are also described using metadata, which is officially termed as a ***manifest***.
- The manifest contains information about the current version of the assembly, culture information (used for localizing string and image resources), and a list of all externally referenced assemblies that are required for proper execution.

Single-File and Multifile Assemblies

If an assembly is composed of a single *.dll or *.exe module, you have a *single-file assembly*. A single-file assembly contains the assembly manifest, the type metadata, the CIL code, and the resources. Figure 1.3 shows a single-file assembly and its contents.

Multifile 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.

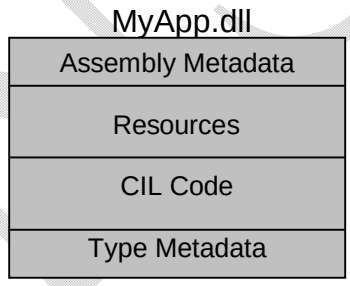


Figure 1.3 *Single-File Assembly*

In other words, multifile assemblies are used when different modules of the application are written in different languages. 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. The multifile assembly shown in Figure 1.4 consists of three files. The MyApp.dll file contains the assembly manifest for the multifile assembly. The MyLib.netmodule file contains the type metadata and the MSIL code but not the assembly manifest. The Employee.gif is the resource file for this multifile assembly.

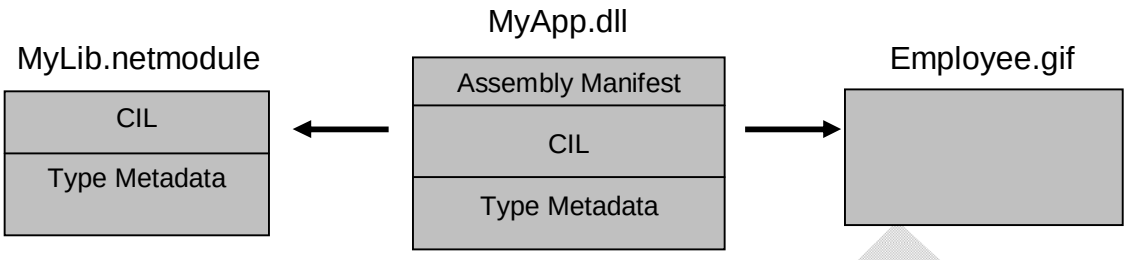


Figure 1.4 *MultiFile Assembly*

Thus, an assembly is really a *logical grouping* of one or more related modules that are intended to be initially deployed and versioned as a single unit.

1.7 The Role of the Common Intermediate Language

- 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# compiler (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

The benefits of compiling source code into CIL rather than directly to a specific instruction set are as given below:

- Language integration: Each .NET-aware compiler produces nearly identical CIL instructions. Therefore, all languages are able to interact within a well-defined binary arena.
- Given that CIL is platform-agnostic, the .NET Framework itself is platform-agnostic. So a single code base can run on numerous operating systems, just like Java.

There is an international standard for the C# language, and a large subset of the .NET platform and implementations already exist for many non-Windows operating systems. In contrast to Java, the .NET allows you to build applications using your language of choice.

1.8 The Role of .NET Type Metadata

- In addition to CIL instructions, a .NET assembly contains full, complete, and accurate metadata.
- The metadata describes each and every type (class, structure, enumeration etc.) defined in the binary, as well as the members of each type (properties, methods, events etc.).
- It is always the job of the compiler (not the programmer) to produce the latest and greatest type metadata.

- As .NET metadata is so careful, assemblies are completely self-describing entities and so .NET binaries have no need to be registered into the system registry.
- Metadata is used by numerous aspects of the .NET runtime environment, as well as by various development tools.
- For example, the IntelliSense feature provided by Visual Studio 2005 is made possible by reading an assembly's metadata at design time.
- Metadata is also used by various object browsing utilities, debugging tools, and the C# compiler itself.
- To be sure, metadata is the backbone of numerous .NET technologies including remoting, reflection, late binding, XML web services, and object serialization.

1.9 The Role of the Assembly Manifest

- The .NET assembly also contains metadata that describes the assembly itself, technically termed a ***manifest***.
- Among other details, the manifest documents all external assemblies required by the current assembly to function correctly, the assembly's version number, copyright information, and so on.
- Like type metadata, it is always the job of the compiler to generate the assembly's manifest.
- The manifest documents the list of external assemblies required by *.exe(via the .assembly extern directive) as well as various characteristics of the assembly itself (version number, module name, and so on).

1.10 Compiling CIL to Platform-Specific Instructions

- Since assemblies contain CIL instructions, rather than platform-specific instructions, CIL code must be compiled before use.
- The entity that compiles CIL code into meaningful CPU instructions is termed a ***just-in-time (JIT) compiler***, which is also known as ***Jitter***.
- 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.

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. In this way, developers can write a single body of code that can be efficiently JIT-compiled and executed on machines with different architectures. Furthermore, as a given Jitter compiles CIL instructions into corresponding machine code, it will cache the results in memory in a manner suited to the target operating system. In this way, if a call is made to a method named PrintDocument(), the CIL instructions are compiled into platform-specific instructions on the first

invocation and retained in memory for later use. Therefore, the next time PrintDocument() is called, there is no need to recompile the CIL.

1.11 Understanding the Common Type System

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

Now, we will discuss the usage of five types defined by CTS.

CTS Class Types

Every .NET-aware language supports, the notion of a *class type*, which is the basis of object-oriented programming (OOP). A class may be composed of any number of members (such as properties, methods, and events) and data points. 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;
        }
}
```

CTS allow a given class to support virtual and abstract members that define a polymorphic interface for derived classes. But CTS classes may only derive from a single base class (multiple inheritance is not allowed fro class).

Following table lists number of characteristics pertaining to class types.

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

The concept of a structure is also formalized under the CTS.

- Structure is a user-defined type (UDT), which can be thought of as a lightweight class type having value-based semantics.
- CTS structures may define any number of *parameterized* constructors.
- CTS 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.***
- Typically, structures are best suited for modeling geometric and mathematical data, and are created in C# using the ***struct*** keyword.

Consider an 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 Types

- *Interfaces* are nothing more than a named collection of abstract member definitions, which may be supported (i.e., implemented) by a given class or structure.
- Interfaces are the only .NET type that does not derive from a common base type.
- This indicates that interfaces are pure protocol and do not provide any implementation.
- On their own, interfaces are of little use. However, when a class or 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 interfaces using a .NET aware programming language, the CTS permits a given interface to derive from *multiple* base interfaces.
- This allows to build some interesting behaviors.
- In C#, interface types are defined using the ***interface*** keyword, for example:

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

CTS Enumeration Types

- *Enumerations* are a handy programming construct that allows you to group name/value pairs under a specific name.
- By default, the storage used to hold each item within enumeration type is a 32-bit integer (System.Int32).
- However, it is possible to alter this storage slot if needed (e.g., when programming for a low-memory device such as a Pocket PC).
- The CTS demands that enumerated types derive 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.
- In C#, enumerations are defined using the keyword ***enum.***

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 type
public enum CharacterType
{
        Wizard = 100,
        Fighter = 200,
        Thief = 300
}
```

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);
```

CTS Type Members

We have seen till now that most types in CTS can take any number of *members*. Formally speaking, a ***type member*** is constrained by the set {constructor, finalizer, static constructor, nested type, operator, method, property, indexer, field, read only field, constant, event}. The CTS defines various “adornments” that may be associated with a given member. For example, each member has a given visibility feature (e.g., public, private, protected). Some members may be declared as abstract to enforce a polymorphic behavior on derived types as well as virtual to define a canned (but overridable) implementation. Also, most members may be configured as static (bound at the class level) or instance (bound at the object level). The construction of type members is examined later in detail.

1.12 Intrinsic CTS Data Types

CTS provide a well-defined set of intrinsic data types used by all .NET aware languages. Although a given language typically has a unique keyword used to declare an intrinsic CTS data type, all language keywords ultimately resolve to the same type defined in an assembly named mscorlib.dll.

Following table lists how key CTS data types are expressed in various .NET languages.

.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.UInt32	UInteger	uint	unsigned int or unsigned long
System.UInt64	ULong	ulong	unsigned __int64
System.Single	Single	float	Float
System.Double	Double	double	Double
System.Object	Object	object	Object^
System.Char	Char	char	wchar_t
System.String	String	string	String^
System.Decimal	Decimal	decimal	Decimal
System.Boolean	Boolean	bool	Bool

1.13 Understanding the Common Language Specification

We know that, different languages express the same programming constructs in unique, language specific terms. For example, in C# you denote string concatenation using the plus operator (+), while in VB .NET we use the ampersand (&). Even when two distinct languages express the same programmatic idiom (e.g., a function with no return value), the chances are very good that the syntax will appear quite different on the surface:

```
' VB .NET method returning nothing.
Public Sub MyMethod()
```

```
        ' Some code...
End Sub

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

Since the respective compilers (like vbc.exe and csc.exe) produce similar CIL instructions, the variation in syntax is minor for .NET runtime. However, languages can also differ with regard to their overall level of functionality. For example, C# may allow to overload some type of operator which VB.NET may not. Given these possible variations, it would be ideal to have a baseline to which all .NET-aware languages are expected to conform.

The Common Language Specification (CLS) is a set of rules that describe 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.

In many ways, the CLS can be viewed as a *subset* of the full functionality defined by the CTS. The CLS is ultimately a set of rules that compiler builders must follow, if they plan their products to function properly 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.

In addition to Rule 1, the CLS defines numerous other rules. For example, the CLS describes how a given language must represent text strings, how enumerations should be represented internally (the base type used for storage), how to define static members, and so on. But the internal understanding of the CTS and CLS specifications is for tool/compiler builders.

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.

1.14 Understanding the Common Language Runtime

Programmatically speaking, the term *runtime* can be understood as a collection of external services that are required to execute a given compiled unit of code. For example, when developers make use of the Microsoft Foundation Classes (MFC) to create a new application, they are aware that their program requires the MFC runtime library (i.e., mfc42.dll). Other popular languages also have a corresponding runtime. VB6 programmers are also tied to a runtime

module (e.g., msvbvm60.dll). Java developers are tied to the Java Virtual Machine (JVM) and so on.

The .NET platform offers a runtime system, which can be described as follows:

- The key difference between the .NET runtime and the various other runtimes is that the .NET runtime provides a single well-defined runtime layer that is shared by *all* languages and platforms that are .NET-aware.
- The root of the CLR is physically represented by a library 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.
- The runtime engine is responsible for a number of tasks.
- First and foremost, it is the entity in-charge of resolving the location of an assembly and finding the requested type within the binary by reading the contained metadata.
- The CLR then lays out the type in memory, compiles the associated CIL into platform-specific instructions, performs any necessary security checks, and then executes the code in question.
- In addition to loading your custom assemblies and creating your custom types, the CLR will also interact with the types contained within the .NET base class libraries when required.

Although the entire base class library has been broken into a number of discrete assemblies, the key assembly is mscorlib.dll. The mscorlib.dll 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. When you build .NET solutions, you automatically have access to this particular assembly.

Figure 1.5 illustrates the workflow that takes place between the source code (which is making use of base class library types), a given .NET compiler, and the .NET execution engine.

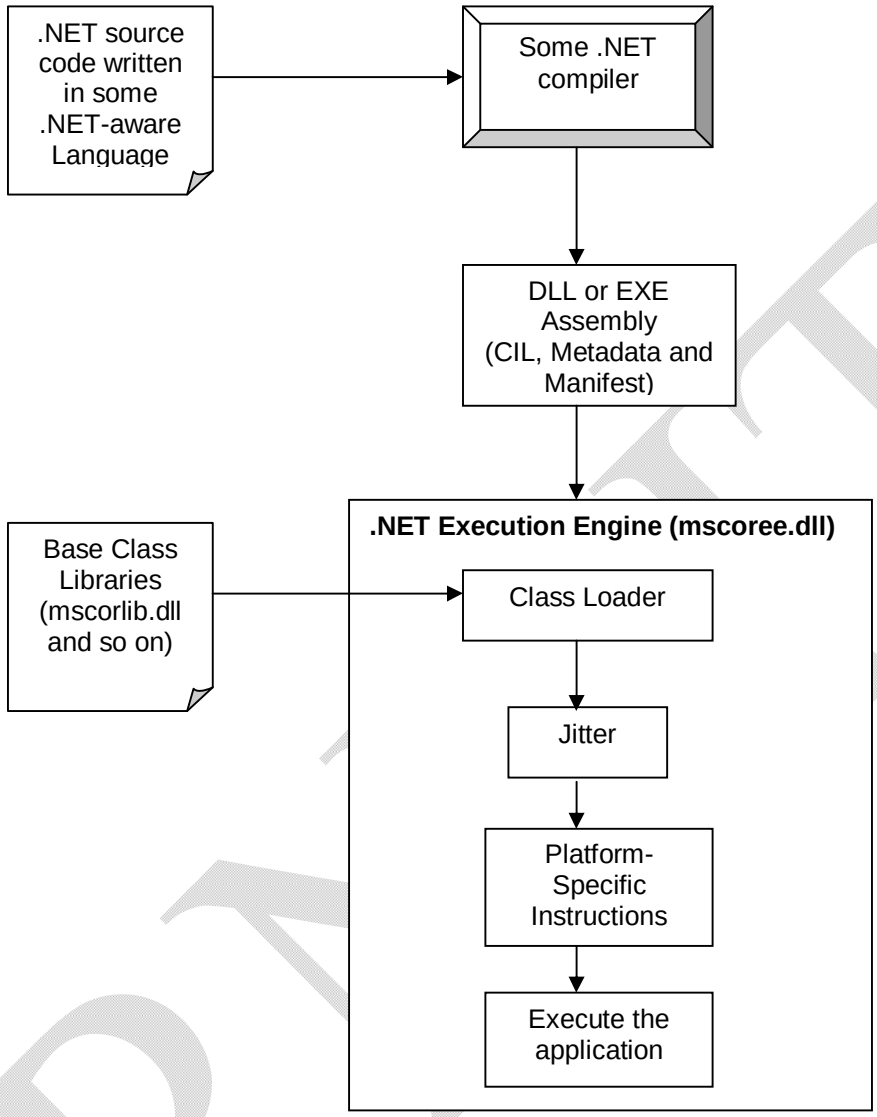


Figure 1.5 *mscoree.dll in action*

1.15 A tour of the .NET Namespaces

- Unlike C, C++ and Java, the C# language does not provide any language-specific code library.
- Instead, C# provides the language-neutral .NET libraries.
- To keep all the types within the base class libraries, the .NET platform makes extensive use of the *namespace* concept.
- A ***namespace*** is a grouping of related types contained in an assembly.
- In other words, ***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.
- It is very important to point out that a single assembly (such as mscorlib.dll) can contain any number of namespaces, each of which can contain any number of types.

The advantage of having namespaces is that, any language targeting the .NET runtime makes use of same namespaces and same types as a C# developer. For example, consider following programs written in C#, VB.NET and MC++.

```
// C# code
using System;
public class Test
{
        public static void Main()
        {
                Console.WrtieLine("Hello World");
        }
}

// VB.NET code
Imports System
Public Module Test
        Sub Main()
                Console.WrtieLine("Hello World")
        End Sub
End Module

// MC++ code
#using <mscorlib.dll>
using namespace System;

void Main()
{
        Console::WrtieLine("Hello World");
}
```

Note that each language is making use of *Console* class defined in the *System* namespace. Apart from minor syntactic variations, three programs look very similar, both physically and logically.

There are numerous base class namespaces within .NET. The most fundamental namespace is *System.* The *System* namespace provides a core body of types that the programmer need to control time. In fact, one cannot build any sort of functional C# application without at least making a reference to the System namespace. Following table shows some of the namespaces:

.NET Namespace	Meaning
System	System contains numerous useful types dealing with intrinsic data, mathematical computations, random number generation, environment variables, and garbage collection, as well as a number of commonly used exceptions and attributes.
System.Collections	These namespaces define a number of stock container objects
System.Collections.Generic	(ArrayList, Queue etc), as well as base types and interfaces that allow you to build customized collections. As of .NET 2.0, the collection types have been extended with generic capabilities.
System.Data System.Data.Odbc System.Data.OracleClient System.Data.OleDb System.Data.SqlClient	These namespaces are used for interacting with databases using ADO.NET.
System.Diagnostics	Here, you find numerous types that can be used to programmatically debug and trace your source code.
System.Drawing System.Drawing.Drawing2D System.Drawing.Printing	Here, you find numerous types wrapping graphical primitives such as bitmaps, fonts, and icons, as well as printing capabilities.
System.IO System.IO.Compression System.IO.Ports	These namespaces include file I/O, buffering, and so forth. As of .NET 2.0, the IO namespaces now include support compression and port manipulation.
System.Net	This namespace (as well as other related namespaces) contains types related to network programming (requests/responses, sockets, end points, and so on).
System.Reflection System.Reflection.Emit	These namespaces define types that support runtime type discovery as well as dynamic creation of types.
System.Runtime. InteropServices	This namespace provides facilities to allow .NET types to interact with “unmanaged code” (e.g., C-based DLLs and COM servers) and vice versa.
System.Runtime. Remoting	This namespace (among others) defines types used to build solutions that incorporate the .NET remoting layer.
System.Security	Security is an integrated aspect of the .NET universe. In the security-centric namespaces you find numerous types dealing with permissions, cryptography, and so on.
System.Threading	This namespace defines types used to build multithreaded applications.
System.Web	A number of namespaces are specifically geared toward the development of .NET web applications, including ASP.NET and XML web services.
System.Windows.Forms	This namespace contains types that facilitate the construction of traditional desktop GUI applications.
System.Xml	The XML-centric namespaces contain numerous types used to interact with XML data.

Accessing a Namespace Programmatically

A namespace is a convenient way to logically understand and organize related types. Consider the System namespace. From programmer's perspective, System.Console represents a class named *Console* that is contained within a namespace called *System*. However, in the eyes of the .NET runtime, it is a single entity named *System.Console*.

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);
                ...
        }
}
```

Remember that both the approaches (a short-hand method with *using* and making use of fully qualified name) results in the *exact* same underlying CIL and has no effect on performance or the size of the assembly.

1.16 Increasing Your Namespace Nomenclature

There are thousands of namespaces in .NET framework. But what makes a namespace unique is that the types it defines are all somehow *semantically related.* So, depending on our requirement, we can make use of only required namespaces. For example, if we are building console applications, then we need not worry about *System.Windows.Forms* and *System.Drawing* namespaces. To know more about specific namespaces, we can use any one of the following:

- .NET SDK online documentation (MSDN – Microsoft Developer Network)
- The ildasm.exe utility
- The Class Viewer Web application
- The wincv.exe desktop application
- The Visual Studio.NET integrated Object Browser

1.17 Deploying the .NET Runtime

The .NET assemblies can be executed only on a machine that has the .NET Framework installed. As an individual who builds .NET software, this should never be an issue, as your development machine will be properly configured at the time you install the freely available .NET Framework 2.0 SDK or a commercial .NET development environments such as Visual Studio 2005.

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. This installation program is included with the .NET Framework 2.0 SDK, and it is also freely downloadable from www.microsoft.com.

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).

Note that if you are building a .NET web application, the end user's machine does not need to be configured with the .NET Framework, as the browser will simply receive generic HTML and possibly client-side JavaScript.

Frequently Asked Questions:

1. Explain the basic building block of .NET framework. (10)
2. Bring out the important differences between single and multifile assemblies. (4)
3. What are namespaces? List and explain the purpose of at least five namespaces. (6)
4. Explain with a neat diagram, the workflow that takes place between your source code, a given .NET complier and the .NET execution engine. (5)
5. What are the key features of C#? (5)
6. Briefly discuss the state of affairs that eventually led to the .NET platform. What is the .NET solution and what C# brings to the table? (10)
7. Explain common type system in detail. (5)
8. Explain the concept of .NET binaries. (4)
9. Explain the role of JIT compiler. (3)
10. Explain the role of CIL and the benefits of CIL in .NET platform? (8)

UNIT 2. Building C# Applications

In this chapter, we will examine the C# compiler, features of Visual Studio IDE and such other basic concepts.

2.1 The Role of the Command-Line Compiler (csc.exe)

There are a number of techniques to compile C# source code. One way is to use the C# command-line compiler, ***csc.exe*** (where csc stands for *C-Sharp Compiler*). This tool is included with the .NET Framework 2.0 SDK. Though we may not build a large-scale application using the command-line compiler, it is important to understand the basics of how to compile *.cs files by hand. Few reasons for having a grip on the process are:

- The most obvious reason is the simple fact that one may not have a copy of Visual Studio 2005. But only free .NET SDK.
- One may plan to make use of automated build tools such as MSBuild or NAnt.
- One may want to understand C# deeply. When you use graphical IDEs to build applications, you are ultimately instructing csc.exe how to manipulate your C# input files. In this light, it is better to see what takes place behind the scenes.
- Another benefit of working with csc.exe is that you become that much more comfortable in manipulating other command-line tools included with the .NET Framework 2.0 SDK. Moreover, a number of important utilities are accessible only from the command line.

Configuring the C# Command-Line Compiler

Before starting the C# command-line compiler, we need to ensure that our development machine recognizes the existence of csc.exe. If the machine is not configured correctly, we have to specify the full path to the directory containing csc.exe before compiling C# files.

We have to set the path by following the steps:

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

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

Note that, the above line may not be exactly the same in every computer. We must enter the current version and location of .NET Framework SDK. To check whether the setting has been done properly or not, open a command prompt and type

csc -? Or csc /?

If settings were correct, we will see a list of options supported by the C# compiler.

Configuring Additional .NET Command-Line Tools

We have to set Path variable with one more line as

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

This directory contains additional command-line tools that are commonly used during .NET development. With these two paths established, we will be able to run any .NET utility from any command window. To confirm this new setting, enter the following command in a command prompt to view the options of the GAC (global assembly cashe) utility, gacutil.exe:

gacutil -?

Note that the above explained method is to compile the C# program in any command prompt. In stead of that, we can directly use the SDK in the Visual Studio Tools menu.

2.2 Building C# Applications Using csc.exe

To build a simple C# application, open a notepad and type the following code:

```
// A simple C# application.
using System;
class Test
{
        public static void Main()
        {
                Console.WriteLine("Hello World!!!");
        }
}
```

Now, save this file in any convenient location as **Test.cs**. Now, we have to provide the option for which type of output file we want. The options for output are given in the following table:

File Output Option	Meaning
/out	This option is used to specify the name of the assembly to be created. By default, the assembly name is the same as the name of the initial input *.cs -file (in the case of a *.dll) or the name of the type containing the program's Main() method (in the case of an *.exe).
/target:exe	This option builds an executable console application. This is the default file output type, and thus may be omitted when building this application type.
/target:library	This option builds a single-file *.dll assembly.
/target:module	This option builds a *module*.Modules are elements of multifile assemblies.
/target:winexe	Although you are free to build Windows-based applications using the /target:exe flag, the /target:winexe flag prevents a console window from appearing in the background.

To compile TestApp.cs, use the command:

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

Most of the C# compilers support an abbreviated version, such as /t rather than /target. Moreover, /target is a default option. So, we can give simply,

csc Test.cs

Now an exe file will be created with the name Test.exe. Now the program can be run by typing –

Test.exe

at the command prompt.

Referencing External Assemblies

Here we will see how to compile an application that makes use of types defined in a separate .NET assembly. Note that mscorlib.dll is *automatically referenced* during the compilation process. To illustrate the process of referencing external assemblies, consider an example –

```
using System;
using System.Windows.Forms;

class Test
{
        public static void Main()
        {
                MessageBox.Show("Hello...");
        }
}
```

Since we have made use of the MessageBox class, we must specify the System.Windows.Forms.dll assembly using the /reference flag which can be abbreviated to /r as –

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

Now, running the application will give the output as –

Compiling Multiple Source Files with csc.exe

When we have more than one *.cs source file, then we can compile them together. For illustration, consider the following example –

File Name: MyTest.cs

```
using System;
using System.Windows.Forms;
```

```
class MyTest
{
        public void display()
        {
                MessageBox.Show("Hello...");
        }
}
```

File Name: Test.cs

```
using System;
class Test
{
        public static void Main()
        {
                MyTest  t = new MyTest ();
                t.display();
        }
}
```

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

```
csc /r:System.Windows.Forms.dll Test.cs MyTest.cs
```

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

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

2.3 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 *.cs input files. To reduce the burden of typing those flags every time, the C# compiler provides ***response files.*** C# response files contain all the instructions to be used during the compilation of a program. These files end in a *.rsp (response) extension.

Consider the following file, which is a response file for the example Test.cs discussed in the previous section.

```
# This is the response file for the Test.exe app
# External assembly references.
```

```
/r:System.Windows.Forms.dll
# output and files to compile (using wildcard syntax).
/target:exe /out:Test.exe *.cs
```

Note that # symbol is used for comment line. If we have saved this file in the same directory as the C# source code files to be compiled, we have to use following command to run the program.

```
csc @Test.rsp
```

If needed, we can specify multiple *.rsp files as input (e.g., csc @FirstFile.rsp @SecondFile.rsp @ThirdFile.rsp). In case of multiple response files, the compiler processes the command options as they are encountered. Therefore, command-line arguments in a latter *.rsp file can override options in a previous response file.

Also note that flags listed explicitly on the command line before a response file will be overridden by the specified *.rsp file. Thus, if we use the statement,

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

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

Note The */reference* flag is cumulative. Regardless of where you specify external assemblies (before, after, or within a response file) the end result is a summation of each reference assembly.

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
```

2.4 Generating Bug Reports

The C# compiler provides a flag named /bugreport to specify a file that will be populated by csc.exe with various statistics regarding current build; including any errors encountered during the compilation. 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. For example, consider the program –

```
using System;
```

```
class Test
{
        public static void Main()
        {
                Console.WriteLine("Hello")          //note that ; is not given
        }
}
```

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.

2.5 Remaining C# Compiler Options

C# compiler has many flags that can be used to control how the .NET assembly to be generated. Following table lists some of the flags and their meaning.

Command Line flags of csc.exe	Meaning
@	Allows to specify a response file used during compilation
/? Or /help	Prints the list of all command line flags of csc.exe
/addmodule	Used to specify the modules to add to a multifile assembly
/baseaddress	Used to specify the preferred base address at which to load a *.dll
/bugreport	Used to build text-based bug reports for the current compilation
/checked	Used to specify whether integer arithmetic that overflows the bounds of the data type will cause an exception at run time
/codepage	Used to specify the code page to use for all source code files in the compilation
/debug	Forces csc.exe to emit debugging information
/define	Used to define preprocessor symbols
/doc	Used to construct an XML documentation file
/filealign	Specifies the size of sections in the output file
/fullpaths	Specifies the absolute path to the file in compiler output
/incremental	Enables incremental compilation of source code files
/lib	Specifies the location of assemblies referenced via /reference

/linkresource	Used to create a link to a managed resource
/main	Specifies which Main() method to use as the program's entry point if multiple Main() methods have been defined in the current *.cs file set.
/nologo	Suppresses compiler banner information when compiling the file
/nostdlib	Prevents the automatic importing of the core .NET library, mscorlib.dll
/noconfig	Prevents the use of *.rsp files during the current compilation
/nowarn	Suppress the compiler's ability to generate specified warnings
/optimize	Enable or disable optimization
/out	Specifies the name of the output file
/recurse	Searches subdirectories for source files to compile
/reference	Used to reference an external assembly
/resource	Used to embed .NET resources into the resulting assembly
/target	Specifies the format of the output file.
/unsafe	Compiles code that uses the C# "unsafe" keyword
/utf8output	Displays compiler output using UTF-8 encoding
/warn	Used to sent warning level for the compilation cycle
/warnaserror	Used to automatically promote warnings to errors
/win32icon	Inserts an .ico file into the output file
/win32res	Inserts a Win32.resource into the output file

2.6 The Command-Line Debugger (cordbg.exe)

The .NET Framework SDK provides a command-line debugger named ***cordbg.exe***. This tool provides dozens of options that allow you to debug your assembly. You may view them by specifying the /? flag:

```
cordbg /?
```

Following table lists some of the flags recognized by cordbg.exe (with the alternative shorthand notation) once you have entered a debugging session.

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

Usually, we will make use of the VS.NET integrated debugger. Hence, the *cordbg.exe* is rarely used in reality.

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

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

This generates a new file named `testapp.pdb`. If you do not have an associated *.pdb file, it is still possible to make use of cordbg.exe; however, you will not be able to view your C# source code during the process (which is typically no fun whatsoever, unless you wish to complicate matters by reading CIL code). Once you have generated a *.pdb file, open a session with cordbg.exe by specifying your .NET assembly as a command-line argument (the *.pdb file will be loaded automatically):

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

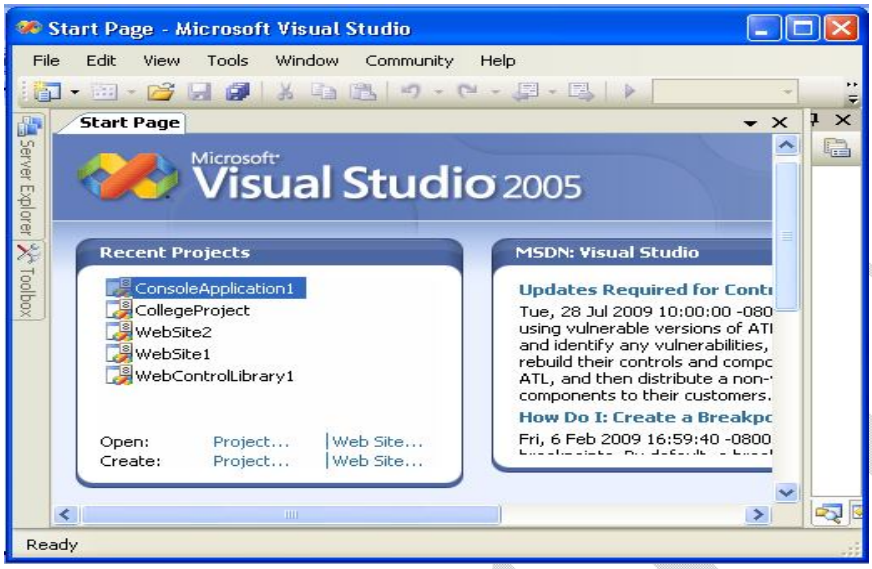
At this point, you are in debugging mode and may apply any number of cordbg.exe flags at the (cordbg) command prompt.

2.7 Using the Visual Studio .NET IDE

Till now, we have discussed how to compile C# programs in command prompt. Visual Studio .NET provides an IDE (Integrated Development Environment) to build applications using any number of .NET-aware languages like C#, VB.NET, J#, MFC etc.

Here, we will examine some of the core features of VS.NET IDE.

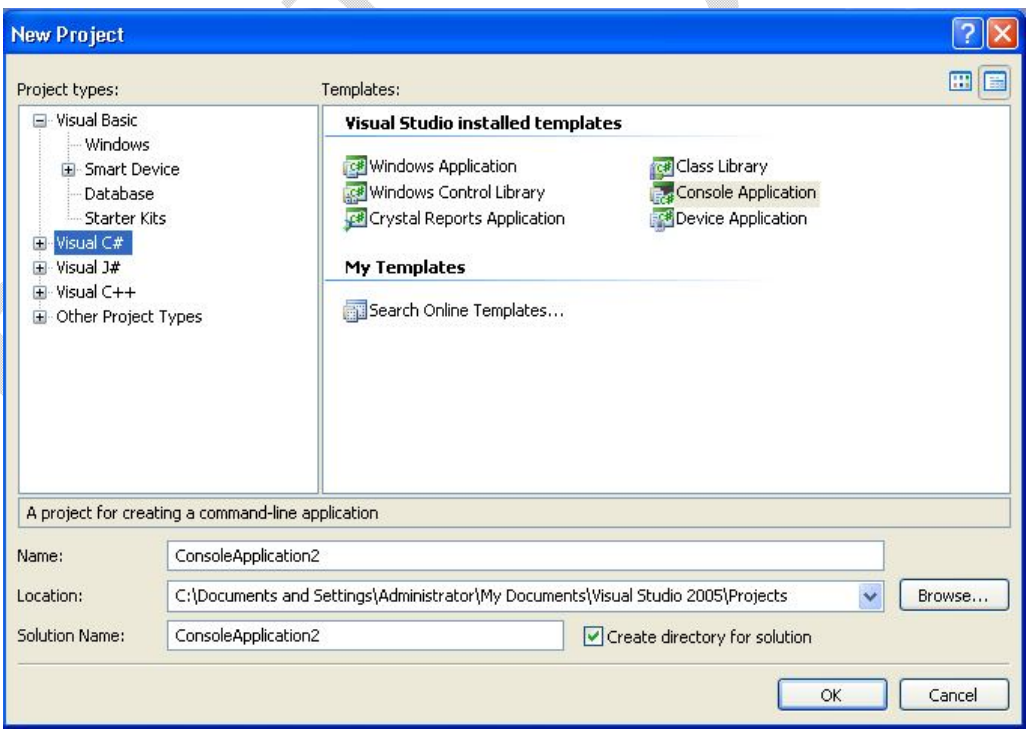
The VS .NET Start Page
When we launch VS .NET, we can see start page as –



This will provide an option to open existing projects or to create new projects.

Creating a VS .NET Project Solution

Once we opt for creating new project, then, we can see the following window:



This window will allow us to choose the language (C#, VB.NET etc), then type of the application (windows, console etc). Then we can provide the name of the application and the location where the application to be stored. Following is a list of project types provided by C#.

Project Type	Meaning
Windows Application	Represents Windows Form application
Class Library	Used to build a single file assembly (*.dll)
Windows Control Library	Allows to build a single file assembly (*.dll) that contains custom Windows Forms Controls. (ActiveX controls)
ASP.NET Web Application	Used to build a web application
ASP.NET Web Service	Used to build a .NET Web Service. Web Service is a block of code, reachable using HTTP requests.
Web Control Library	Used to build customized Web controls. These GUI widgets are responsible for emitting HTML to a requesting browser.
Console Application	Just like command window.
Windows Services	Used to build NT/2000 services. Services are background worker applications that are launched during OS boot process.

2.8 C# Preprocessor Directives

Like C and C++, C# provides various symbols to interact with the compilation process. But, the term *preprocessor* doesn't exactly mean what it implies in C or C++. There is no separate preprocessing step in C#. Rather, preprocessor directives are processed as a part of the lexical analysis phase of the compiler. However, the syntax of C# preprocessor is similar to that of C and C++. Following is a list of preprocessors supported by C#.

C# Preprocessor Symbol	Meaning
#define, #undef	Used to define and un-define conditional compilation symbol.
#if, #elif, #else, #endif	Used to conditionally skip sections of source code.
#line	Used to control the line numbers emitted for errors and warnings
#error, #warning	Used to issue errors and warnings for the current build
#region, #endregion	Used to explicitly mark sections of source code. Under VS.NET, regions may be expanded and collapsed within the code window. Other IDEs (including text editors) will ignore these symbols.

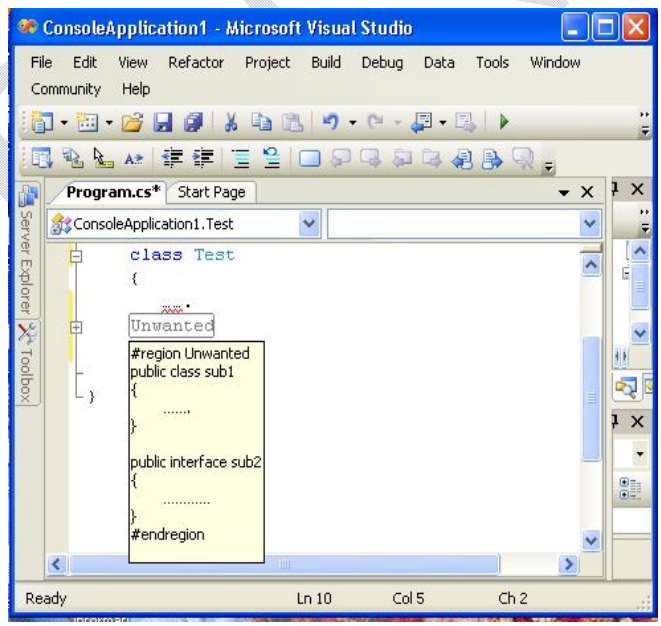
Specifying Code Regions

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
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:
#warning and #error directives allows to instruct 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:

```
#define MAX
using System;
class Test
{
      public static void Main()
      {
        #if(MAX)
         #warning MAX is defined by me!!!
                ……..
        #endif
      }
}
```

When we compile, the program will give warning message –

```
Test.cs(7,21): warning CS1030: #warning: 'MAX is defined by me!!!'
```

Altering Line Numbers:
The directive #line is rarely used in any project. This directive allows to alter 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:

```
#define MAX
using System;
class Test
{
      public static void Main()
      {
        #line 300                                   //next line will take the number 300
         #warning MAX is defined by me!!!
                  ……….
         #line default                              // default numbering will continue
      }
}
```

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.

2.9 An Interesting Aside: The System.Environment Class

The System.Environment class allows to obtain a number of details regarding the context of the operating system hosting .NET application by using various static members. Consider the following example –

```
using System;
class Test
{
      public static void Main(string[] args)
      {
         Console.WriteLine("OS: {0}", Environment.OSVersion);
         Console.WriteLine("Current Directory: {0}", Environment.CurrentDirectory);

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

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

The Output will be (depending on OS and version of .NET, it may vary) –

```
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:\
Version of .NET: 2.0.50727.3082
```

Frequently Asked Questions:

1. List and explain the various output options available with C# complier. Also illustrate with an example, how do you compile multiple source files? (08)
2. What are response files? Explain with an example. (05)
3. How do you generate bug reports? Illustrate with an example. (05)
4. What are C# preprocessor directives? Explain. (10)
5. Explain the use of System.Environment class with a program. (05)

UNIT 3. C# Language Fundamentals

In this chapter, we will examine the data types in C#, decision and iteration constructs, boxing and unboxing, role of System.Object and so on.

3.1 The Anatomy of a Basic C# Class

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

Program 3.1 Test.cs

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

Every executable C# application must contain a class defining a *Main()* method, which is used to signify the entry point of the application. *Main()* is used with the public and static keywords. The public members are accessible from other types, while static members are scoped at the class level (rather than the object level) and can thus be invoked without the need to first create a new class instance (or object). In addition to the public and static keywords, this Main() method has a single parameter, which is an array of strings (string[] args). This parameter may contain any number of incoming command-line arguments.

The program *Test.cs* makes use of the Console class, which is defined within the System namespace. WriteLine() is a static member of Console class, which is used to pump a text string to the standard output.

NOTE: We can use Console.ReadLine() to ensure the command prompt launched by Visual Studio 2005 remains visible during a debugging session until we press the Enter key.

Variations on the Main() Method

The *Main()* function can be take any of the following forms based on our requirement:

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

// Integer return type, no arguments.
public static int Main()
{
        //some code
        //return a value to OS
}
```

NOTE: The Main() method may also be defined as private. Doing so ensures other assemblies cannot directly invoke an application's entry point. Visual Studio 2005 automatically defines a program's Main() method as private. Obviously, your choice of how to construct Main() will be based on two questions. First, do you need to process any user-supplied command-line parameters? If so, they will be stored in the array of strings. Second, do you want to return a value to the system when Main() has completed? If so, you need to return an integer data type rather than void.

Processing Command-Line Arguments

Assume that we need some arguments to be passed as input to the Main() funtion before starting the program execution. This is done through command-line parameters. Consider an example –

Program 3.2

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

Here, we are checking to whether the array of strings contains some number of items using the *Length* property of *System.Array*. We can provide command line arguments while running the program as –

```
Test.exe Good Bad Ugly
```

As an alternative to the standard *for* loop, we may iterate over incoming string arrays using the C# *foreach* keyword. For example –

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

It is also possible to access command-line arguments using the static *GetCommand-LineArgs()* method of the *System.Environment* type. The return value of this method is an array of strings. The first index identifies the current directory containing the application itself, while the remaining elements in the array contain the individual command-line arguments. If we are using this technique, we need not define the Main() method with string array parameter. Consider an example –

```
public static int Main()          //no arguments
{
    // Get arguments using System.Environment.
    string[] Args = Environment.GetCommandLineArgs();
    Console.WriteLine("Path to this app is: {0}", Args[0]);

    for(int i=1; i<Args.Length;i++)
        Console.WriteLine("Arguments are: {0}", Args[i]);
}
```

3.2 Creating Objects: Constructor Basics

All object-oriented (OO) languages make a clear distinction between classes and objects. A *class* is a definition (or, if you will, a *blueprint*) for a user-defined type (UDT). An *object* is simply a term describing a given instance of a particular class in memory. In C#, the *new* keyword is used to create an object. Unlike other OO languages (such as C++), it is not possible to allocate a class

type on the stack; therefore, if you attempt to use a class variable that has not been "new-ed," you are issued a compile-time error. Thus the following C# code is *illegal*:

```
using System;
class Test
{
        public static int Main(string[] args)
        {
                Test c1;
                c1.SomeMethod();        //error!! Must use "new"
                …
        }
}
```

To illustrate the proper procedures for object creation, consider the following code:

```
using System;
class Test
{
        public static int Main(string[] args)
        {
                // You can declare and create a new object in a single
                line…
                Test c1 = new Test();

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

The *new* keyword will calculate the correct number of bytes for the specified object and acquires sufficient memory from the managed heap. Here, we have allocated two objects c1 and c2, each of which points to a unique instance of *Test* type. Note that C# object variables are really a r*eference* to the object in memory, not the actual object itself. Thus, c1 and c2 each reference a distinct *Test* object allocated on the managed heap.

In the previous program, the objects have been constructed using the *default constructor*, which by definition never takes arguments. Every C# class is automatically provided with a free default constructor, which you may redefine if needed. The default constructor ensures that all member data is set to an appropriate default value. But in C++, un-initialized data gets garbage value.

Usually, classes provide additional constructors also. Using this, we can initialize the object variables at the time of object creation. Like in Java and C++, in C# constructors are named

identically to the class they are constructing, and they never provide a return value (not even void).

Consider the following example –

Program 3.3

```
using System;
class Test
{
        public string userMessage;

        // Default constructor.
        public Test()
        {
                Console.WriteLine("Default constructor called!");
        }

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

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

                // Call parameterized constructor.
                Test c2;
                c2 = new Test ("Hello World");
                Console.WriteLine("Value of userMessage: {0}", c2.userMessage);
                Console.ReadLine();
                return 0;
        }
}
```

NOTE:

1. Technically speaking, when a type defines identically named members (including constructors) that differ only in the number of—or type of—parameters, the member in question is *overloaded*.

2. As soon as we define a custom constructor for a class type, the free default constructor is removed. If we want to create an object using the default constructor, we need to explicitly redefine it as in the preceding example.

Is That a Memory Leak?
Note that, in previous program, we have not written any code to explicitly destroy the c1 and c2 references. In C++, if we do like this, it will lead to memory leak. But, the .NET garbage collector frees the allocated memory automatically, and therefore C# does not support a *delete* keyword.

3.3 The Composition of a C# Application

We have seen in the previous example that the Main() function creates instances of the very own class in which it is defined. However, a more natural way to write an application using distinct classes is as shown in the following example. In OO terminology, this way of writing an application is known as ***Separation of concerns.***

Program 3.4 MyTest.cs

```
using System;
class Test
{
        public Test()
        {
                Console.WriteLine("Default constructor called!");
        }
        public void disp()
        {
                Console.WriteLine("Hi");
        }
}

class MyTest
{
        public static void Main(string[] args)
        {
                Test c1 = new Test ();
                c1.disp();
        }
}
```

The type (like class, structure etc) containing Main() function (that is, entry point for the application) is called as the ***application object*****.** Every C# application will be having one application object and numerous other types. On the other hand, we can create an application object that defines any number of members called from the Main() method.

3.4 Default Assignments and Variable Scope

All intrinsic .NET data types have a default value. When we create custom types, all member variables are automatically assigned to their respective default values. Consider an example –

Program 3.5

```
class Test
{
        public int a;               //default value is 0
        public byte b;              //default value is 0
        public char c;              //default value is null
        public string s;            //default value is null

        public static void Main()
        {
                Test t=new Test();
                Console.WriteLine("{0}, {1}, {2}, {3}", a, b, c, s);
        }
}
```

All the variables will be initialized to their default values when we start debugging. However, we can not expect the same within method scope. For example, the following program will generate an error.

```
class Test
{
        public static void Main()
        {
                int a;                  //need to assign some value to 'a'
                Console.WriteLine("{0}", a);        //Error: Unassigned local variable 'a'
        }
}
```

NOTE: There is an exception for the mandatory assignment of a local variable. If the local variable is used as an ***output parameter***, then it need not be initialized before use. Output parameters are discussed later in detail. However, note that, they are used to receive the value from a function.

3.5 The C# Member Variable Initialization Syntax

Consider a class (or any other type) having more than one constructor with different set (may be different in type or number) of arguments. Now, if any member variable needs to be initialized with same value in all the situations, we need to write the code repeatedly in all the constructors. For example,

```
class Test
{
        private int a;

        Test()
        {
                a=5;
        }

        Test(int k)
        {
                a=5;
                .......
        }

        Test(string s)
        {
                a=5;
                .......
        }
}
```

Though the above code is valid, there is redundancy in code. We can write a function for such initialization like –

```
class Test
{
        private int a;

        public void init()
        {
                a=5;
        }

        Test()
        {
                init();
        }

        Test(int k)
        {
                init()
```

```
            .....
        }

        Test(string s)
        {
            init()
            .....
        }
    }
```

Here we have function call instead of assignment. Thus, redundancy still exists. To avoid such situation, C# provides the initialization of member variables directly within a class as shown –

```
class Test
{
    private int a =5;
    private string s= "Hello";
    ..............
}
```

This facility was not available in C++. Note that, such an initialization will happen before the constructor gets called. Hence, if we provide any other value to a variable through constructor, the previous member assignment will be overwritten.

3.6 Basic Input and Output with the Console Class

In many of the programs what we have seen till now, made use of System.Console class. As the name suggests, Console class is defined in System namespace and it encapsulates input, output and error stream manipulations. This class is widely used for console applications but not for Windows or Web applications.

There are four important static methods in Console class:

- Read() Used to capture a single character from the input stream.
- ReadLine() Used to receive information from the input stream until the enter key is pressed.
- Write() This method pumps text to the output stream without carriage return (enter key)
- WriteLine() This pumps a text string including carriage return to the output stream.

Following table gives some important member of Console class:

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 WindowTop WindowLeft	These properties control the dimensions of the console in relation to the established buffer.

To illustrate the working of Console class methods, consider the following example –

Program 3.6

```
using System;
class BasicIO
{
        public static void Main()
        {
                Console.Write("Enter your name: ");
                string s = Console.ReadLine();
                Console.WriteLine("Hello, {0} ", s);
                Console.Write("Enter your age: ");
                s = Console.ReadLine();
                Console.WriteLine("You are {0} years old", s);
        }
}
```

The output would be –

```
Enter your name: Ramu
Hello, Ramu
Enter your age: 25
You are 25 years old
```

Formatting Textual Output

In the previous examples, we have seen numerous occurrences of the tokens {0}, {1} etc. embedded within a string literal. .NET introduces a new style of string formatting. A simple example follows –

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

```
        int i = 90;
        double d = 9.99;
        bool b = true;

        Console.WriteLine("Int is: {0}\nDouble is: {1}\nBool is: {2}", i, d, b);
    }
```

The first parameter to WriteLine() represents a string literal that contains optional placeholders designated by {0}, {1}, {2}, and so forth (curly bracket numbering always begins with zero). The remaining parameters to WriteLine() are simply the values to be inserted into the respective placeholders

Note that WriteLine() has been overloaded to allow the programmer to specify placeholder values as an array of objects. Thus, we can represent any number of items to be plugged into the format string as follows:

```
// Fill placeholders using an array of objects.
object[] stuff = {"Hello", 20.9, 1, "There", "83", 99.99933} ;
Console.WriteLine("The Stuff: {0} , {1} , {2} , {3} , {4} , {5} ", stuff);
```

It is also permissible for a given placeholder to repeat within a given string. For example, if we want to build the string "9, Number 9, Number 9" we can write

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

Note If any mismatch occurs between the number of uniquely numbered curly-bracket placeholders and fill arguments, an exception viz. FormatException is popped up at runtime.

.NET String Formatting Flags

If we require more detailed formatting, each placeholder can optionally contain various format characters (in either uppercase or lowercase), as shown –

C# Format Character	Meaning
C or c	Used to format currency. By default, the flag will prefix the local cultural symbol (a dollar sign [$] for U.S. English). However, this can be changed using a 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	Used for exponential notation.
F or f	Used for fixed-point formatting.
G or g	Stands for *general*. This character can be used to format a

	number to fixed or exponential format.
N or n	Used for basic numerical formatting (with commas).
X or x	Used for hexadecimal formatting. If you use an uppercase X, your hex format will also contain uppercase characters.

These format characters are suffixed to a given placeholder value using the colon token (e.g., {0:C}, {1:d}, {2:X}, and so on). Consider an example:

Program 3.7

```
class Test
{
    public static void Main(string[] args)
    {
        Console.WriteLine("C format: {0:C}", 99989.987);
        Console.WriteLine("D9 format: {0:D9}", 99999);
        Console.WriteLine("E format: {0:E}", 99999.76543);
        Console.WriteLine("F3 format: {0:F3}", 99999.9999);

        Console.WriteLine("N format: {0:N}", 99999);
        Console.WriteLine("X format: {0:X}", 99999);
        Console.WriteLine("x format: {0:x}", 99999);
    }
}
```

The output would be –

```
C format: $99,989.99
D9 format: 000099999
E format: 9.999977E+004
F3 format: 100000.000
N format: 99,999.00
X format: 1869F
x format: 1869f
```

Note that the use of .NET formatting characters is not limited to console applications. These same flags can be used within the context of the static String.Format() method. This can be helpful when we need to create a string containing numerical values in memory for use in any application type (Windows Forms, ASP.NET, XML web services, and so on):

Program 3.8

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

```
            string Str;

            Str = String.Format("You have {0:C} in your account", 99989.987);
            Console.WriteLine(Str);
        }
    }
```

The output would be –

```
You have $99,989.99 in your account
```

3.7 Understanding Value Types and Reference Types

Like any programming language, C# defines a number of keywords that represent basic data types such as whole numbers, character data, floating-point numbers, and Boolean values. These intrinsic types are fixed constants. That is, all .NET-aware languages understand the fixed nature of these intrinsic types, and all agree on the range it is capable of handling.

A .NET data type may be ***value-based*** or ***reference-based***. Value-based types, which include all numerical data types (int, float, etc.) as well as enumerations and structures, are allocated *on the stack*. So, value types can be quickly removed from memory once they are out of the defining scope:

```
public void SomeMethod()
{
    int i = 0;
    Console.WriteLine(i);
}       // 'i' is removed from the stack here
```

When we assign one value type to another, a member-by-member (or a bit-wise) copy is achieved by default. In terms of numerical or Boolean data types, the only “member” to copy is the value of the variable itself:

```
public void SomeMethod()
{
    int i = 99;
    int j = i;
    j = 8732;     // j will be 8732 and I remains to be 99
}
```

This example may not seem special. But, now we will have a look at other value types like structures and enumerations. Structures provide a way to achieve the benefits of object orientation (i.e., encapsulation) while having the efficiency of stack-allocated data. Like a class, structures can take constructors (only parameterized constructors) and define any number of members. All structures are implicitly derived from a class named **System.ValueType**. The

purpose of System.ValueType is to "override" the virtual methods defined by System.Object to support value-based v/s reference-based semantics. In fact, the instance methods defined by System.ValueType are identical to those of System.Object.

Consider an example to illustrate the working of value types –

Program 3.9 MyClass.cs

```
struct MyStruct
{
        public int x;
}

class MyClass
{
        public static void Main()
        {
                // the keyword new is optional while creating structures.
                MyStruct ms1 =new MyStruct();

                ms1.x = 100;

                MyStruct ms2 = ms1;

                Console.WriteLine("ms1.x = {0}", ms1.x);     //100
                Console.WriteLine("ms2.x = {0}", ms2.x);     //100

                ms2.x = 900;
                Console.WriteLine("ms1.x = {0}", ms1.x);     //100
                Console.WriteLine("ms2.x = {0}", ms2.x);     //900
        }
}
```

The Output will be –

```
ms1.x = 100
ms2.x = 100
ms1.x = 100
ms2.x = 900
```

Note that, to allocate a structure type, we can use the 'new' keyword. This will create an impression that memory is allocated in heap. But, this is not true. CLR makes the programmer to feel everything is an object and new value types. However, when the runtime encounters a type derived from System.ValueType, stack allocation is achieved.

In the above program, we have created a variable ms1 and then it is assigned to ms2. Since MyStruct is a value type, we will have two copies of the MyStruct type on the stack, each of which can be independently manipulated. Therefore, when we change the value of ms2.x, the value of ms1.x is unaffected.

On the other hand, the reference types (classes) are allocated on the managed heap. These objects stay in memory until the .NET garbage collector destroys them. By default, assignment of reference types results in a new reference to the *same* object on the heap. To illustrate, let us change the above example:

Program 3.10

```
class MyClass            //struct has been changed to class
{
        public int x;
}

class MyClass1
{
        public static void Main()
        {
                // the keyword new is compulsory now.
                MyClass mc1 =new MyClass ();

                mc1.x = 100;

                MyClass mc2 = mc1;

                Console.WriteLine("mc1.x = {0}", mc1.x);     //100
                Console.WriteLine("mc2.x = {0}", mc2.x);     //100

                ms2.x = 900;
                Console.WriteLine("mc1.x = {0}", mc1.x);     //900
                Console.WriteLine("mc2.x = {0}", mc2.x);     //900
        }
}
```

The Output will be –

```
mc1.x = 100
mc2.x = 100
mc1.x = 900
mc2.x = 900
```

In this example, we have two references pointing to the same object on the managed heap. Therefore, when we change the value of x using the mc2 reference, mc1.x reports the same value.

Value Types Containing Reference Types

Assume that we have a reference (class) type *MyClass* containing a constructor to assign value to a member variable of type string. Assume also that we have included object of this class within a structure *MyStruct*. We have included the Main() function in another class and the total application is as given below –

Program 3.11

Name of file: Test.cs

```
using System;

class MyClass
{
        public string s;
        public MyClass(string str)        //constructor for class
        {
                s=str;
        }
}

struct MyStruct
{
        public MyClass mc;
        public int a;

        public MyStruct(string str)       //constructor for structure
        {
                mc=new MyClass(str);      //calling constructor of class
                a=10;
        }
}

class Test
{
        public static void Main(string[] args)
        {
                MyStruct ms1=new MyStruct("Initial Value");
```

```
            ms1.a=15;

            MyStruct ms2=ms1;

            ms2.mc.s="New Value!!!";
            ms2.a=20;
            Console.WriteLine("ms1.mc.s is {0}", ms1.mc.s);
            Console.WriteLine("ms2.mc.s is {0}", ms2.mc.s);
            Console.WriteLine("ms1.a is {0}", ms1.a);
            Console.WriteLine("ms2.a is {0}", ms2.a);
        }
    }
```

The Output would be –

```
ms1.mc.s is New Value!!!
ms2.mc.s is New Value!!!
ms1.a is 15
ms2.a is 20
```

Now compare the programs **3.9, 3.10 and 3.11.**

- In **Program 3.9,** we have used the structure, which is of value type. So, when we use the statement

  MyStruct *ms2= ms1;*

  there will be separate memory allocation for two objects (Memory will be allocated from stack). Thus, the assignment statement *ms2=ms1* will copy the contents of *ms1* to the corresponding variables of *ms2*. Since memory allocation is different for these two objects, the change in one object doesn't affect the other object.

- In **Program 3.10,** we have used the class, which is of reference type. So, when we use the statement

  MyClass *mc2 = mc1*,

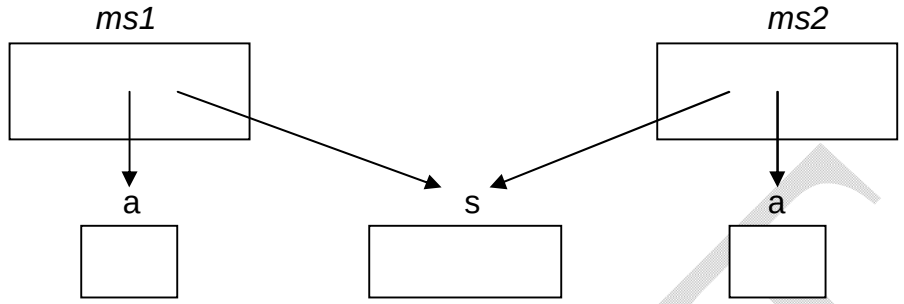
  mc2 will be just reference to *mc1* sharing the same memory location in the heap. Also, the contents of *mc1* will be copied to the corresponding variables of *mc2*. Since memory location is same for both the objects, the change in one object will affect the other object.

- In **Program 3.11**, we have a value type *MyStruct* containing an object of reference type *MyClass*. The object *ms1* of *MyStruct* contains an object *mc* of *MyClass* and an integer variable *a*. Here, *a* will be allocated memory from stack and *mc* will get memory from heap. Now, when we use the statement

  MyStruct ms2=ms1;

  the object *ms2* gets created. Note that, *ms1.a* and *ms2.a* will have different memory allocation as *a* is member of structure. But, *ms1.mc.s* and *ms2.mc.s* will point to same

location in heap memory as *s* is basically a member of class. This can be diagrammatically represented as –



Thus, when a value type contains other reference types, assignment results in a copy *of the references*. In this way, we have two independent structures *ms1* and *ms2* each of which contains a reference pointing to the same object in memory (i.e., a "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, which we will see later in detail.

Value and Reference Types: Final Details

Following table summarizes the core distinctions between value types and reference types.

Intriguing Question	Value Type	Reference Type
Where is this type allocated?	Allocated on the stack	Allocated on the managed heap.
How is a variable represented?	Value type variables are local copies.	Reference type variables are pointing to the memory occupied by the allocated instance.
What is the base type?	Must derive from System.ValueType.	Can derive from any other type (except System.ValueType), as long as that type is not "sealed"
Can this type function as a base to other types?	No. Value types are always sealed and cannot be extended.	Yes. If the type is not sealed, it may function as a base to other types.
What is the default parameter passing behavior?	Variables are passed by value (i.e., a copy of the variable is passed into the called function).	Variables are passed by reference (e.g., the address of the variable is passed into the called function).
Can this type override System.Object.Finalize()?	No. Value types are never placed onto the heap and therefore do not need to be finalized.	Yes, indirectly
Can we define constructors?	Yes, but the default constructor is reserved	Yes
When do variables of this type die?	When they fall out of the defining scope.	When the managed heap is garbage collected.

Despite their differences, value types and reference types both have the ability to implement interfaces and may support any number of fields, methods, overloaded operators, constants, properties, and events.

3.8 The Master Class: 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
{
        …
}
```

But, internally, it means that,

```
class Test : System.Object
{
        …
}
```

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

```
// The 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);
        }
}
```

Following table shows some of the functionality provided by each instance-level method.

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 is overridden to return true only if the objects being compared have the same internal state values. Note that if you override Equals(), you should also override GetHashCode().
GetHashCode()	This method returns an integer that identifies a specific object in memory. If you intend your custom types to be contained in a System.Collections.Hashtable type, you are well-advised to override the default implementation of this member.
GetType()	This method returns a System.Type object that fully describes the details of the current item. In short, this is a Runtime Type Identification (RTTI) method available to all objects.
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. This method can be overridden by a subclass to return a tokenized string of name/value pairs that represent the object's internal state, rather than its fully qualified name.
Finalize()	This protected method (when overridden) is invoked by the .NET runtime when an object is to be removed from the heap (during garbage collection).
MemberwiseClone()	This protected method exists 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. (Example, Program 3.11)

The Default Behavior of System.Object

To illustrate some of the default behavior provided by the System.Object base class, consider the following example –

Program 3.12

```
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!");
        }
}
```

The Output would be –

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

We can notice from the above program that 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. p2 is also of type Person, however, we are not creating a *new* instance of the Person class, but

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 our applications, 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 example to override System.Object.ToString().

Program 3.13

```
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());
        }
}
```

The Output would be –

```
        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 (Discussed later in detail).

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 –

Program 3.14

```
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("Shyam", "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))      //comparison based on content but not on reference
            Console.WriteLine("p4 and p2 are same");
          else
            Console.WriteLine("p4 and p2 are not same");
        }
      }
```

The Output would be –

```
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 is suitable for use in hashing algorithms and data structures such as a hash table.
- 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.
- Furthermore, the .NET Framework does not guarantee the default implementation of the GetHashCode() method, and the value it returns will be the same between different versions of the .NET Framework.
- So, the default implementation of this method must not be used as a unique object identifier for hashing purposes.
- If we place a custom (user-defined) object into a System.Collections.Hashtable type, its Equals() and GetHashCode() members will determine the correct type to return from the container.
- So, user-defined types should redefine the hashing algorithm used to identify itself within such a type.

There are many algorithms that can be used to create a hash code. In the simplest case, an object's hash value will be generated by taking its data into consideration and building a unique numerical identifier for the type. For example, SSN will be unique for each individual. So, we can override GetHashCode() method as below –

Program 3.15

```
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("Shyam", "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");
    }
  }
```

The output would be –

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

3.9 The System Data Types (and C# Aliases)

Every intrinsic C# data type is an alias to an existing type defined in the System namespace. 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(all types excluding interfaces) derive from object	The base class of all types in the .NET universe.

The relationship between the core system types is shown in the figure 3.1. From this diagram, 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, our user-defined type (class, enumeration, structure etc) can be understood by any language in .NET framework.

Experimenting with the System Data Types

From the Figure 3.1, it is clear that C# data types are alias names for a related structure in the System namespace, and hence derived from System.ValueType. The purpose of System.ValueType is to override the virtual methods defined by System.Object to work with value-based v/s reference-based semantics.

Since data types are value types, the comparison of two variables will be based on their internal value, but not the reference:

```
System.Int32 a=20;
int b=20;
if(a==b)                    //comparison based on value
    Console.WriteLine("Same!!");
```

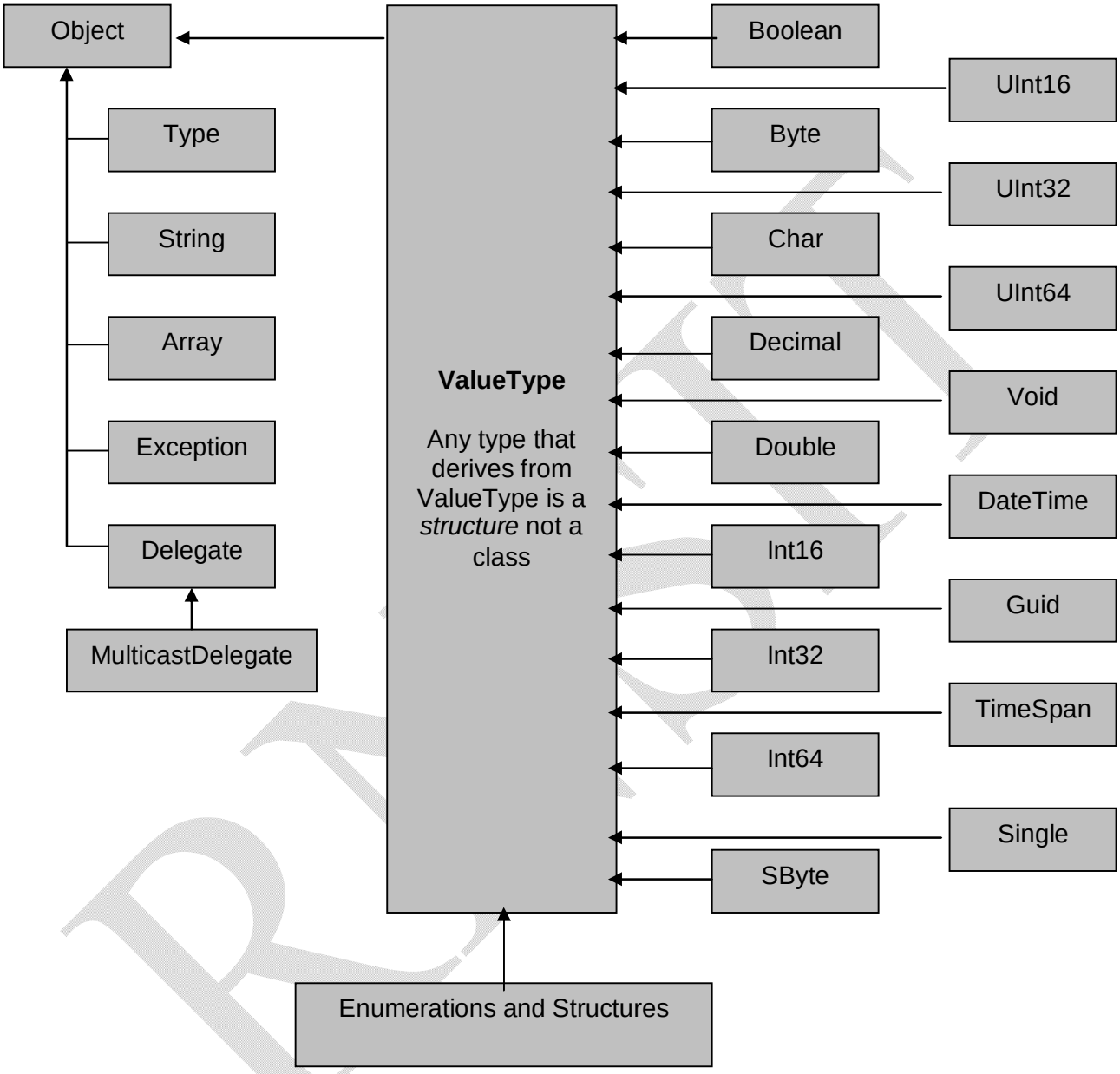


Fig. 3.1 The hierarchy of System Types

Basic Numerical Members

To understand the numerical types in C#, consider the following example –

Program 3.16

```
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

          Console.WriteLine("double.Epsilon: {0}", double.Epsilon);
                                                  //4.94065645841247E-324
          Console.WriteLine("double.PositiveInfinity: {0}", double.PositiveInfinity);
                                                                  //Infinity
          Console.WriteLine("double.NegativeInfinity: {0}", double.NegativeInfinity);
                                                                  //-Infinity
          Console.WriteLine("double.MaxValue: {0}", double.MaxValue);
                                                      // 1.79769313486232E+308
          Console.WriteLine("double.MinValue: {0}", double.MinValue);
                                                      //-1.79769313486232E+308
        }
}
```

Members of System.Boolean

In C#, for Boolean data, we have to assign any one value *true* or *false*. We can not use the numbers like -1, 0, 1 etc. as we do in C++. The usage of bool data is shown in Program 3. 17.

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 –

Program 3.17

```
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
        }
}
```

Parsing Values from String Data

.NET data types provide the ability to generate a variable of their underlying type given a textual equivalent. This technique is called as *parsing*. This is helpful when we need to convert user input data into a numerical value. Consider the following program –

Program 3.18

```
using System;
class Test
{
        public static void Main()
        {
                bool b=bool.Parse("True");
                Console.WriteLine("Value of bool is:{0}", b);          //True
```

```
            double d=double.Parse("99.457");
            Console.WriteLine("Value of double is:{0}", d);      //99.457

            int i=int.Parse("8");
            Console.WriteLine("Value of int is:{0}", i);          //8

            char c=char.Parse("w");
            Console.WriteLine("Value of char is:{0}", c);         //w
        }
}
```

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

- We know that .NET defines two broad categories of 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 can be defined as 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.

  For example,

```
int p=20;

object ob=p;          //box the value type p into an object reference
```

- The operation just opposite to boxing is called as ***unboxing.***
- 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 –

Program 3.19

```
using System;

class Test
{
        public static void MyFunc(object ob)
        {
                Console.WriteLine(ob.GetType());
                Console.WriteLine(ob.ToString());
                Console.WriteLine(((int)ob).GetTypeCode());
        }

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

The output would be –

```
System.Int32
20
Int32
```

NOTE:

1. Boxing and unboxing takes some processing time. So, it must be used only when needed.
2. 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. We will study this later in-detail.

3.11 Defining Program Constants

C# provides a *const* keyword to defined variables with a fixed, unalterable value. The value of a constant data is computed at compile time and hence, a constant variable can not be assigned to

an object reference (whose value is computed at runtime). That is, boxing is not possible for constant variables. We can define either class-level constants or local-level (within a method) constants. For example,

Program 3.20

```
using System;

abstract class ConstClass
{
        public const int p=10;
        public const string s="I am a constant";
}

class Test
{
        public const int x=5;

        public static void Main()
        {
                const int y=20;
                Console.WriteLine("{0}, {1}, {2}, {3}", ConstClass.p, ConstClass.s, x, y);

        }

}
```

The output would be –

```
10, I am a constant, 5, 20
```

3.12 C# Iteration Constructs

C# provides following iteration constructs:

- *for* Loop
- *foreach/in* Loop
- *while* Loop
- *do/while* Loop

The *for* Loop

The *for* loop in C# is just similar to that in C, C++ or Java. This loop will allow to repeat a set of statements for fixed number of times. For example –

```
for(int i=0; i<10;i++)
        Console.WriteLine("{0}", i);            //0 to 9 will be printed.
```

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

The *foreach/in* Loop

This loop is used to iterate over all items within an array. The following example shows how each element of an integer array is considered within a loop.

Program 3.21

```
using System;
class Test
{
    public static void Main()
    {
        int[] arr=new int[]{5, 32, 10, 45, 63};

        foreach(int i in arr)
            Console.WriteLine("{0}",i);
    }
}
```

Apart from iterating over simple arrays, *foreach* loop is used to iterate over system supplied or user-defined collections. This will be discussed in later chapters.

The *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,

```
string s;
while((s=Console.ReadLine())!=null)
    Console.WriteLine("{0}", s);
```

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");
```

3.13 C# Control Flow Constructs

There are two control flow constructs in C# viz. *if/else* and *switch/case*. The *if/else* works only on boolean expressions. So we can not use the values 0, 1 etc. within *if* as we do in C/C++. Thus, the *if* statement in C# typically involve the relational operators and/or conditional operators.

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

Conditional Operator	Meaning
&&	AND
||	OR
!	NOT

The general form of *if* can be given as –

```
if(condition)
{
        //true block
}
else
{
        //false block
}
```

The general from of *switch* can be given as –

```
switch(variable/expression)
{
        case val1:   //statements;
                     break;
        case val2:   //statements;
                     break;
        --------------
        --------------
        case val2:   //statements;
                     break;
}
```

3.14 Complete set of C# 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	=, *=, -=, +=, /=, %=, <<=, >>=, &=, ^=, |=

The relational operator ***is*** is used to verify at runtime whether an object is compatible with a given type or not. The ***as*** operator is used to downcast between types. We will discuss these later in detail. 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.*

3.15 Defining Custom Class Methods

In C#, every data and a method must be a member of a class or structure. That is, we can not have global data or method. The methods in C# 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 in C# 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.

3.16 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,

Program 3.22

```
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

Normally, we will define a set of data members for a class. Then every object of that class will have separate copy of each of those data members. For example,

```
class Test
{
        public int p;
}
-------
------
Test t1=new Test();
t1.p=10;
Test t2= new Test();
t2.p=15;
```

Here, the objects *t1* and *t2* will be having separate copy of the variable *p*.

On the other hand, *static* data is shared among all the objects of that class. That is, for all the objects of a class, there will be only one copy of *static* data. For example,

Program 3.23

```
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
        }
}
```

3.17 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	Reference to a parameter is passed to a method. (call-by-reference)
params	This modifier will allow to send many number of parameters 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*. So, the changes made for parameters within a method will not affect the actual parameters of the calling method. Consider the following example –

Program 3.24

```
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);
        }
}
```

The output would be –

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

The *out* Keyword

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*. The usage of *out* can be better understood by the following example –

Program 3.25

```
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);    //z=25
        }
    }
```

The *out* parameter is certainly useful when we need more values to be returned from a method. Consider the following example –

Program 3.26

```
using System;

class Test
{
    public static void MyFun(out int x, out string s)
    {
        x=5;
        s="Hello, how are you?";
    }

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

        MyFun(out a, out str);
        Console.WriteLine("a={0}, str={1}", a, str);
    }
}
```

The output would be –

a=5, str="Hello, how are you?

The C# *ref* Keyword

Whenever we want the changes made in method to get affected in the calling method, then we will go for call by *ref*. Following are the 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.

Program 3.27

```
using System;

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

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

The output would be –

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

The C# *params* Keyword

The *params* keyword of C# allows us to send many numbers of arguments as a single parameter. To illustrate the use of *params*, consider the following example –

Program 3.28

```
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);
        }
}
```

The output would be –

```
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, as shown in the following program –

Program 3.29

```
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);
        }
    }
```

The output would be –

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

Passing Reference Types by Value and Reference

Till now, we have seen how to pass a parameter to methods by value and by using *ref*. In the previous examples, we have passed *value type* variables as parameters to methods (Just recollect that there are two types of variables *value type* like int, char, string, structure etc. and *reference type* like class, delegates [Section 3.7]). Now we will see what happens when *reference type* variables (i.e. objects of class) are passed as parameters. Consider the program –

Program 3.30

```
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("Shyamu", 25);
    }

    public static void CallByRef(ref Person p)
    {
```

```
            p.age=55;
            p=new Person("Shyamu", 20);
        }

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

        public static void Main()
        {
            Person p1=new Person("Ramu", 30);
            p1.disp();
            CallByVal(p1);
            p1.disp();
            CallByRef(ref p1);
            p1.disp();
        }
    }
```

The output would be –

```
Ramu    30
Ramu    66
Shyamu  20
```

In the Main() function, we have created an object *p1* of *Person* class. Memory will be allocated to *p1* from heap as *p1* is of reference type (object of a class). Now, the display will be –

```
Ramu    30
```

Now we are passing *p1* to the function using call-by-value method. But, by nature, *p1* is of reference type. So, in the receiving end, the object *p* takes the reference of *p1*. That means, the memory location for both *p1* and *p* will be same. Thus, the statement

```
p.age = 66;
```

will affect the original object *p1*. And hence the output will be –

```
Ramu    66
```

But, when we try to allocate new memory for *p*, the compiler will treat *p* as different object and new memory is allocated from heap. Now onwards, *p* and *p1* are different.

Next, we are passing *p1* using the modifier *ref*. This is nothing but call-by-reference. Here also, the receiving object *p* will be a reference to *p1*. Since the parameter passing technique used is

call-by-reference, as per the definition, any changes in method should affect the calling method. Thus, the statement

```
p= new Person("Shyamu", 20);
```

will affect the original object *p1* and hence the output is –

```
Shyamu  20
```

NOTE: The important rule to be remembered is: **"If a class type (reference type) is passed by reference, the called method will change the values of the object's data and also the object it is referencing".**

3.18 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]{"Ramu", "Shymu"};
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 example –

Program 3.31

```
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);
        }
    }
```

The output would be –

```
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. For example –

Program 3.32

```
using System;
class Test
{
    public static void Main()
    {
        int[,] arr=new int[2,3]{{5, 7, 0}, {3,  1, 8}};
        int sum=0;

        for(int i=0;i<2; i++)
            for(int j=0;j<3;j++)
                sum=sum+arr[i,j];
```

```
            Console.WriteLine("Sum is {0}", sum);

        }
    }
```

The output would be –

```
    Sum is 24
```

Jagged array contain some number of inner arrays, each of which may have unique size. For example –

Program 3.33

```
using System;
class JaggedArray
{
    public static int[][] JArr=new int[3][];

    public static void Main()
    {
        int m, n, p, sum=0;

        Console.WriteLine("Enter the sizes for 3 inner arrays:");
        m=int.Parse((Console.ReadLine()).ToString());
        n=int.Parse((Console.ReadLine()).ToString());
        p=int.Parse((Console.ReadLine()).ToString());

        JArr[0]=new int[m];
        JArr[1]=new int[n];
        JArr[2]=new int[p];

        Console.WriteLine("Enter the elements for array:");
        for(int i=0;i<3;i++)
            for(int j=0;j<JArr[i].Length;j++)
            {
                JArr[i][j]=int.Parse((Console.ReadLine()).ToString());
                sum=sum+JArr[i][j];
            }

        Console.WriteLine("\nThe Sum is: {0}",sum);
    }
}
```

The output would be –

```
Enter the sizes for 3 inner arrays:
2       3       2
Enter the elements for array:
1       2       3       4       5       6       7
The Sum is: 28
```

The System.Array Base Class

Every array in C# is derived from the class System.Array. This class defines number of methods to work with arrays. Few of the methods are given below –

Member	Meaning
BinarySearch()	This static 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 static method sets a range of elements in the array to empty values (0 for value types; null for reference types).
CopyTo()	This method is used to copy elements from the source array into the destination array.
Length	This read-only property is used to determine the number of elements in an array.
Rank	This property returns the number of dimensions of the current array.
Reverse()	This static method reverses the contents of a one-dimensional array.
Sort()	This method sorts a one-dimensional array of intrinsic types. If the elements in the array implement the IComparer interface, 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 –

Program 3.34

```
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]);
        }
    }
```

The Output would be –

```
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
```

3.19 String Manipulation in C#

Till now, we have used *string* key word as a data type. But truly speaking, *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 property returns the length of the current string.
Contains()	This method is used to determine if the current string object contains a specified string.
Concat()	This static method of the String class 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 static method is used to format a string literal using other primitives (i.e., numerical data and other strings) and the {0} notation examined earlier in this chapter.
Insert()	This method is used to receive a copy of the current string that contains newly inserted string data.
PadLeft() PadRight()	These methods return copies of the current string that has been padded with specific data.
Remove() Replace()	Use these methods to receive a copy of a string, with modifications (characters removed or replaced).
Substring()	This method returns a string that represents a substring of the current string.
ToCharArray()	This method returns a character array representing the current string.
ToUpper() ToLower()	These methods create a copy of a given string in uppercase or lowercase.

Consider the following example –

Program 3.35

```
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("Cotains 'is'?: {0}", s1.Contains("is"));

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

The output would be –

```
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 sChar 11 is t
Char 12 is r Char 13 is l Char 14 is n       Char 15 is g

Cotains '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). For console applications, this can be an audio clue to the user.
\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
\u	Inserts a Unicode character 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 our literals. Consider the following example –

Program 3.36

```
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);
        }
}
```

The output would be –

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

Using *System.Text.StringBuilder*

In the previous examples, we tried to change the content of strings using various methods like *Replace(), ToUpper()* etc. But, the value of a string cannot be modified once it is established. 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 example –

Program 3.37

```
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);

        }
}
```

The output would be –

```
Hello World
HELLO WORLD
```

3.20 C# 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 –

Program 3.38

```
    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
        }
}
```

The System.Enum Base Class

The C# enumerations are derived from System.Enum class. This base class defines some methods for working with enumerations.

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.

Program 3.39

```
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");
    }
}
```

The output would be –

```
        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
```

3.21 Defining Structures in C#

In C#, structures behave similar to class, except that memory structures will be allocated in stack area, whereas for class memory will be allocated from heap area. Structures can have member data, member methods, constructors (only parameterized) and they can implement interfaces. Structure in C# 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 example –

Program 3.40

```
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, "Ramu");
                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();
        }
    }
```

The output would be –

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

3.21 Defining Custom Namespaces

In the programs we discussed till now, we have used namespaces like *System*, *System.Text* etc. These are existing namespaces in the .NET framework. 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 like –

```
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 and 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
{
                -------
}
```

Frequently Asked Questions:

1. What do you understand by 'params' method of parameter passing? Give an example. (6)
2. What is boxing and unboxing? Explain with an example for each. (6)
3. Explain the C# static methods and static data with suitable examples. (10)
4. Write a program to illustrate the difference between passing reference types by reference and by value. (8)
5. Using the methods in System.String class, design a C# method which replaces all occurrences of the word "computer" with "COMPUTER". (6)

   Program:

```
using System;
class Test
{
        public static void Main()
        {
                string s1="my computer is good computer";
                string s2;
                s2=s1.Replace("computer", "COMPUTER");
                Console.WriteLine("{0}",s2);    //my COMPUTER is good COMPUTER
        }
}
```

6. What is the difference between System.String and System.Text.StringBuilder? Explain with relevant code some of the features of StringBuilder class. (5)
7. Write a C# program to design a structure Student<USN, Name, Marks, Branch>. Here, Branch is of type Enum with members MCA, MBA, MTech. Add appropriate constructor and also a method to hike the marks by 10% to only MCA students. Show creation of some Student objects and the way to call these methods. (12)

```
using System;

public enum Branch
{
        MCA,
        MBA,
        MTech
}

struct Student
{
        string USN;
        string Name;
        double marks;
        Branch b;
```

```
        public Student(string usn, string name, double m, Branch br)
        {
                USN=usn;
                Name=name;
                marks=m;
                b=br;
        }
        public void Hike_Marks()
        {
                if(b==Branch.MCA)
                {
                        marks=marks + 0.1 * marks;
                }
        }
        public void disp()
        {

                Console.WriteLine("{0} \t {1} \t\t {2} \t\t {3}", USN, Name, marks, b);
        }

        public static void Main()
        {
                Student[] s=new Student[4];
                s[0]=new Student("1RN07MCA01","Abhishek",75.0,Branch.MCA);
                s[1]=new Student("1RN07MCA55","Ramu",78.0,Branch.MCA);

                s[2]=new Student("1RN06MBA01","Shamu",72.0,Branch.MBA);
                s[3]=new Student("1RN08MTec01","Rani",75.0,Branch.MTech);

                for(int i=0;i<4;i++)
                        s[i].Hike_Marks();

                Console.WriteLine("***********Student Details *************");
                Console.WriteLine("USN \t \tName \t\t Marks \t\t Branch");

                for(int i=0;i<4;i++)
                        s[i].disp();
        }
}
```

The output would be –

```
**********Student Details *************
USN             Name        Marks     Branch
1RN07MCA01      Abhishek     82.5      MCA
1RN07MCA55      Ramu         85.8      MCA
1RN06MBA01      Shamu        72        MBA
1RN08MTec01     Rani         75        MTech
```

8. Explain various method parameter modifiers used in C#. (5)
9. Design a C# class called Matrix containing an integer matrix as a member and methods to
   (i) read the elements of the matrix from the keyboard
   (ii) find the product of two matrices
   (iii) to print the elements in the matrix form.

   Write a Main() method to call all these methods and to print all the three matrices. (10)
   (Lab Program)
10. Write a program to count the number of objects created for a class. (08)

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

        public Test()
        {
                count++;
        }
        public static void Main()
        {
                Test t1=new Test();
                Test t2=new Test();
                Console.WriteLine("Number of objects={0}", count);          //2
                Test t3=new Test();
                Console.WriteLine("Number of objects={0}", count);          //3
        }
}
```

11. Write a C# application which defines a class Shape, with four data members length, breadth, height and radius, appropriate constructors and methods to calculate volume of cube, cone and sphere. Also write ShapeApp, which creates these 3 objects i.e. cube, cone and sphere using appropriate constructors and calculates their volume with the help of above class methods. (10)

```
using System;

class Shape
{
        double length, breadth, height, radius;
        const double pi=3.1416;
        double volume;

        public Shape(double l, double b, double h)    //for cube
        {
                length=l;
                breadth=b;
                height=h;
        }

        public Shape(double r)          //for sphere
        {
                radius = r;
        }

        public Shape(double r, double h)       //for cone
        {
                radius=r;
                height=h;
        }

        public double Cube_Vol()
        {
                volume = length* breadth * height;
                return volume;
        }
        public double Sphere_Vol()
        {
                volume=(4/3)*pi*Math.Pow(radius,3);
                return volume;
        }

        public double Cone_Vol()
        {
                volume=(1.0/3)*pi*Math.Pow(radius,2) * height;
                return volume;
        }
}
```

```
class ShapeApp
{
        public static void Main()
        {
                Shape s1=new Shape(5.0);      //Sphere
                Shape s2=new Shape(3, 4,5);   //Cube
                Shape s3=new Shape(2, 4);     //Cone

                Console.WriteLine("Volume of Sphere= {0}", s1.Sphere_Vol());
                Console.WriteLine("Volume of Cube= {0}", s2.Cube_Vol());
                Console.WriteLine("Volume of Cone= {0}", s3.Cone_Vol());
        }
}
```

The output would be –

```
Volume of Sphere= 392.7
Volume of Cube= 60
Volume of Cone= 16.7552
```

12. What is the output printed by the following code segment and why? (6)

```
struct Point
{
        public int x, y;
        public Point(int x, int y)
        {
                this.x=x ;
                this.y=y ;
        }
}
Point p=new Point(10,10) ;
object obj=p ;
p.x=20;
Console.WriteLine("p.x={0}", p.x);                    //20
Console.WriteLine("obj.x={0}", ((Point)obj).x) ;      //10
```

UNIT 4. Object-Oriented Programming with C#

In this chapter we will discuss basics of OOPs and OOPs concepts like encapsulation, inheritance and polymorphism.

4.1 Formal Definition of the C# Class

Class is a basis of OOPs. A class can be defined as a user-defined data type (UDT) that is composed of data (or attributes) and functions (or methods) that act on this data. In OOPs, we can group data and functionality into a single UDT to model a real-world entity.

A C# class can define any number of *constructors*. Constructor is a special type of method called automatically when object gets created. They are used to provide initial values for some attributes. The programmer can define default constructor to initialize all the objects with some common state. Custom or parameterized constructors can be used to provide different states to the objects when they get created. The general form of a C# class may look like –

```
class class_name
{
        //data members
        //constructors
        //methods
}
```

Understanding Method Overloading

A class can have more than one method with a same name only if number and/or type of parameters are different. Such methods are known as ***overloaded*** methods. For example –

```
// Example for overloaded constructors
class Employee
{
        public Employee()       //default constructor
        {    }

        public Employee(string name, int EmpID, float BasicSal)
        {
                //some code
        }
}

// overloaded member-methods: number of arguments are different
class Triangle
{
```

```
        public float Draw(float height, float base)          //two arguments
        {
                //some code
        }
        public float Draw(float sideA, float sideB, float sideC)    //3 arguments
        {
                //some code
        }
}
```

```
// overloaded member-methods: type of arguments are different
class Shape
{
        public float Area(float height, float base)
        {
                //some code
        }

        public int Area(int a, int b)
        {
                //some code
        }
}
```

```
// overloading can not be done only based on return-type
class Shape
{
        //error!!!
        public float Area(float height)   //one parameter of type float
        {
                //some code
        }

        public int Area(float a )         //one parameter of type float
        {
                //some code
        }
}
```

Self-Reference in C#

The key-word ***this*** is used to make reference to the ***current object*** i.e. the object which invokes the function-call. For example –

```
class Employee
{
        string name;
        int eId;

        public Employee(string name, int EmpID)
        {
                this.name=name;
                this.eId=EmpID;
        }
}
```

Note that the static member functions of a class can not use *this* keyword as static methods are invoked using class name but not the object.

Forwarding Constructor Calls using "*this*"

Using *this* keyword, we can force one constructor to call another constructor during constructor call. This will help us to avoid redundancy in member initialization logic.

Consider that there is a class called *IDGenerator* containing a static method to generate *employee id* based on some pre-defined criteria like department and designation of employee. The application may be having two situations: (a) provide an employee id at the time of object creation, and (b) the employee id must be generated from *IDGenerator* class. But the parameter like name of the employee is provided during object creation itself. In such a situation, to avoid code-redundancy, we may use *this* keyword to forward one constructor call to other as shown below –

```
class IDGenerator
{
        static int id;

        public static int GetEmpID()
        {
                //code for generating an employee id viz id
                return id;
        }
}

class Employee
{
        string name;
        int eID;
        float BasicSal;
```

```
        public Employee(string n, int EmpID, float b)
        {
                this.name=n;            //either use this
                this.eID=EmpID;
                BasicSal=b;             //or not use this
        }

        /* GetEmpID() method is called first to generate employee-id and
        then the construct call is forwarded to the above constructor */

        public Employee(string n): this(n, IDGenerator.GetEmpID(), 0.00)
        {       }
        public static void Main()
        {
                //direct call for constructor with three arguments
                Employee e1=new Employee("Ramu", 111, 12000.00);

                /* call for constructor with single arguments which in-turn
                calls constructor with three arguments later
                */
                Employee e2=new Employee("Shyamu");
                ------------------
        }
}
```

In the above example, if we would have not forwarded the constructor call, then we would need to write redundant code in the second constructor like –

```
public Employee(string n)
{
        this.name=n;
        this.eID=IDGenerator.GetEmpID();
        this.BasicSal=0.00;
}
```

Thus, using *this* keyword to forward constructor call, we are avoiding code redundancy.

4.2 Defining the "Default Public Interface" of a Type

The term *default public interface* refers to the set of public members (data or method) that are directly accessible from an object. That means, the default public interface is any item declared in the class with the keyword *public*. In C#, the default public interface of a class may be any of the following:

- *Methods* : Named units that model some behavior of a class
- *Properties* : Accessor and mutator functions
- *Public data* : A data member which is public (though it is not good to use, C# provides if programmer needs)

Apart from above items, default public interface may include user defined event, delegates and nested types.

Specifying Type Visibility: Public and Internal Types

We know that any member of a class can be declared for its level of visibility (access specifier) using the keyword *public, private, protected, internal* and *protected internal*. Just like members, the type (class, structure, interface, delegate, enumeration) itself can be specified for its level of visibility.

The ***method/member visibility*** is used to know which members can be accessed from an object of the type, where as, the ***type visibility*** is used to know which parts of the system can create the object of that type.

A non-nested C# type can be marked by either *public* or *internal* and nested types can be specified with any of *public, private* and *internal.* The public types can be created by any other objects within the same assembly or by other external assemblies. But, the internal types can be created by the types within the same assembly and are not accessible outside the assembly. By default, the visibility level of a class is *internal*. For example,

```
//this class can be used outside this assembly also
public class Test
{
        //body of the class
}

//this class can be accessed only within its assembly
internal class Test
{
        //body of the class
}

Or
class Test          //by default, it is internal
{
        //body of the class
}
```

4.3 Recapping the Pillars of OOP

All the object oriented languages possess three principals (considered to be pillars of OOP) viz.

- Encapsulation : How the languages hide an object's internal implementation?
- Inheritance : How the languages promote code reuse?
- Polymorphism : How the languages let the programmer to treat related objects in a similar way?

Here, we will discuss the basic role of each of these pillars and then move forward for detailed study.

Encapsulation Services

Encapsulation is the ability to hide unnecessary implementation details from the object user. For example, assume we have created a class named DBReader having methods *open()* and *close()*:

```
DBReader d=new DBReader();
d.open("C:\MyDatabase.mdf");
……..
d.close();
```

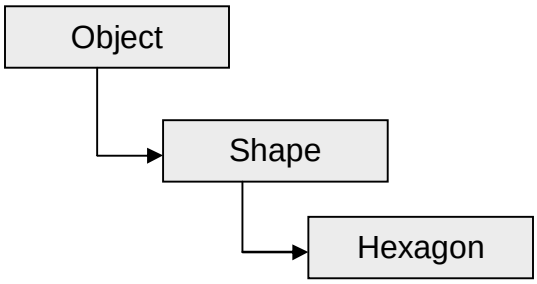
The class DBReader has encapsulated the inner details of locating, loading, manipulating and closing data file. But, the object user need not worry about all these.

Closely related to the notion of encapsulation is ***data hiding.*** We do this by making data members as *private*. The private data can be modified only through the public member functions of that class.

Inheritance: The *is-a* and *has-a* Relationships

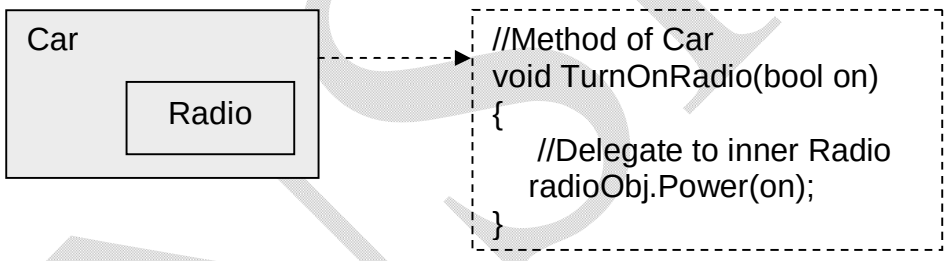
Inheritance is the ability to create new class definitions based on existing class definitions. In other words, inheritance allows us to extend the behavior of base class by inheriting core functionality into a derived class.

For example, we know that System.Object is the topmost class in .NET. We can create a class called Shape which defines some properties, fields, methods and events that are common to all the shapes. The Hexagon class extends Shape and inherits properties of Shape and Object. It contains properties of its own. Now, we can say Hexagon is a Shape, which is an object. Such kind of relationship is called as ***is-a*** relationship and such inheritance is termed as ***Classical inheritance.***



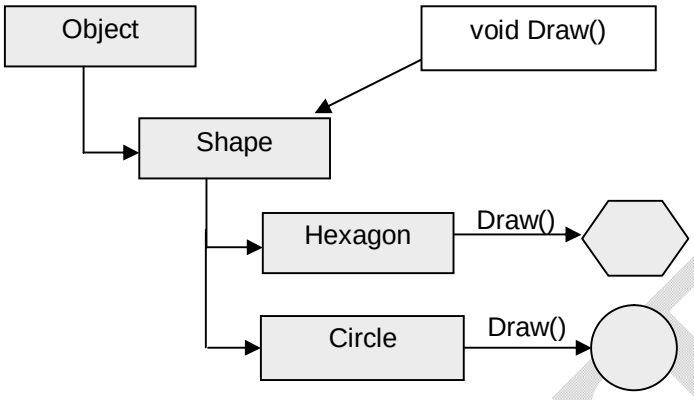
There is another type of code reuse in OOP viz. the ***containment/delegation model*** or ***has-a*** relationship. This type of reuse is not used for base class/child class relationships. Rather, a given class can define a member variable of other class and make use of that either fully or partly.

For example, a Car can contain Radio. The containing type (Car) is responsible for creating the inner object (Radio). If the Car wishes to make the Radio's behavior accessible from a Car instance, it must provide some set of public functions that operate on the inner type.



Polymorphism: Classical and Ad Hoc

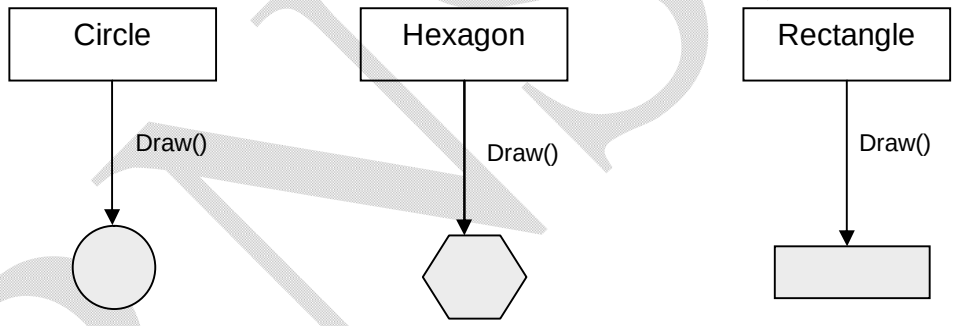
Polymorphism can be of two types viz. ***classical*** and ***Ad Hoc***. A ***Classical polymorphism*** takes place in the languages that support classical inheritance. The base class can define a set of members that can be overridden by a subclass. When subclass overrides the behavior defined by a base class, they are essentially redefining how they respond to the same message. For example, assume Shape class has defined a function named Draw() without any parameter and returning nothing. As every shape has to be drawn in its own manner, each subclass like Hexagon, Circle etc. can redefine the method Shape() as shown –



Classical polymorphism allows a base class to enforce a given behavior on all subclasses.

Ad hoc polymorphism allows objects that are ***not*** related by classical inheritance to be treated in a similar manner, provided that every object has a method of the exact signature. Languages that support ad hoc polymorphism possess a technique called ***late binding*** to find the underlying type of given object at runtime.

Consider the following situation:



Note that the three classes Circle, Hexagon and Rectangle are not from same base. Still each class supports an identical Draw() method. This is possible by implementing Draw() method on generic types. When a particular type is used for calling Draw(), it will act accordingly.

4.4 The First Pillar: C#'s Encapsulation Services

The concept of encapsulation says that the object's data should not be directly accessible from a method. If the data is to be manipulated, it has to be done indirectly using accessor (get) and mutator(set) method. C# provides following two techniques to manipulate private data members –

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

It is good programming practice to make all the data members or fields of a class as private. This kind of programming is known as black box programming.

Enforcing Encapsulation Using Traditional Accessors and Mutators

If we want to access any private data, we can write a traditional ***accessor*** (get method) and when we want to provide value to data, we can write a ***mutator*** (set method). For example,

```
class Emp
{
        string Name;
        public string GetName()          //accessor
        {
                return Name;
        }
        public void SetName(string n)          //mutator
        {
                Name=n;
        }
}
class Test
{
        public static void Main()
        {
                Emp p=new Emp();
                p.SetName("Ramu");
                Console.WriteLine("{0}",p.GetName());
        }
}
```

In this example, using the public methods, we could able access private data.

Another Form of Encapsulation: Class Properties

Apart from traditional accessors and mutators, we can make use of properties provided by .NET class (and structures, interfaces). Properties resolve to a pair of hidden internal methods. The user need not call two separate methods to get and set the data. Instead, user is able to call what appears to be a single named field. For illustration, consider the following example –

```
using System;
class Emp
{
        string Name;
```

```
        public string EmpName          // EmpName is name of the property
        {
                get
                {
                        return Name;            // Name is field-name
                }
                set
                {
                        Name=value;             // value is a keyword
                }
        }
}

class Test
{
        public static void Main()
        {
                Emp p=new Emp();
                p.EmpName="Ramu";           //use name of the property

                Console.WriteLine("{0}",p.EmpName);
        }
}
```

A C# property is composed using a get block (accessor) and set block (mutator). The *value* keyword represents the right-hand side of the assignment. Though, *value* is an object, the underlying type of the object depends on which kind of data it represents. For example, in the previous program, the property *EmpName* is operating on a private string, which maps to System.String. Unlike traditional accessors and mutators, the properties make our types easier to manipulate.

Properties are able to respond to the intrinsic operators in C#. For example,

```
using System;
class Emp
{
        int Sal;

        public int Salary
        {
                get
                {
```

```
                return Sal;
            }
            set
            {
                Sal=value;
            }
        }
    }
    class Test
    {
        public static void Main()
        {
            Emp p=new Emp();
            p.Salary=12000;
            p.Salary++;
            Console.WriteLine("{0}",p.Salary);          //12001
            p.Salary -=400;
            Console.WriteLine("{0}",p.Salary);          //11601
        }
    }
```

Read-only and Write-only Properties

In some of the situations, we just want to set a value to a data member and don't want to return it. In such a situation, we can omit *get* block. A property with only a *set* block is known as ***write-only*** property. In the same manner, we can just have a *get* block and omit *set* block. A property with only a *get* block is known as ***read-only*** property. For example,

```
class Emp
{
    string SSN, name;
    int EmpID;

    public Emp(string n, int id)
    {
        name=n;
        EmpID=id;
    }
    public string EmpSSN          //read-only property
    {
        get{ return SSN;}
    }
}
```

Understanding static properties

C# supports *static* properties. Note that, static members of a class are bound to class but not for objects. That is, to access static members, we need not create an object of the class. The same rule will apply to static properties too. For example,

```
class Emp
{
        static string CompanyName;

        public static string Company
        {
                get{  return CompanyName;    }
                set{  CompanyName=value;     }
        }

        public static void Main()
        {
                Emp.Company="RNSIT";          //use class name
                Console.WriteLine("We are at {0}", Emp.Company);
        }
}
```

Understanding static Constructors

As we have seen, static properties can be used to set and get values for static members. Assume a situation, where we need to just set a value commonly for all the objects. Writing a static write-only property with a set-block and then assigning value is quite time-consuming. For this purpose, C# provides another method for doing so, through *static* constructors. For example,

```
class Emp
{
        static string CompanyName;
        static Emp()
        {
                CompanyName="RNSIT";
        }
        public static void Main()
        {
                Console.WriteLine("We are at {0}", Emp.Company);
        }
}
```

Now, all the objects of *Emp* class will be set the value *"RNSIT"* for the member *CompanyName* automatically as soon as the objects gets created.

4.5 Pseudo-Encapsulation: Creating Read-only Fields

Just like read-only properties, we have a notion of read-only fields. Read-only fields offer data preservation via the keyword ***readonly***. The readonly field can be given a value through assignment only at the time of declaration or as a part of constructor.

```
class Student
{
        public readonly int Sem=5;          //assignment during declaration
        public readonly string USN;
        string name;

        public Student(string n, string usn)
        {
                name=n;
                USN=usn;          //assignment through constructor
        }

        public static void Main()
        {
                Student s1=new Student("Abhishek", "1RN07MCA01");
                s1.Sem= 3;  //error
                s1.USN="1RN07MCA02";              //error
                ……..
        }
}
```

NOTE:

- The keyword ***readonly*** is different from ***const***.
- The *const* fields can be assigned a value at the time of declaration only and assignment is not possible there-after.
- So, *const* fields will have same value through out the program, as it is ***compile-time* constant.**
- On the other hand, *readonly* fields can be assigned a value through constructor also.
- So, they can have different values based on constructor call.
- For example –

```
class Student
{
    public readonly int Sem=5;      //make Sem as 5 for all objects
    public readonly string USN;
    string name;
```

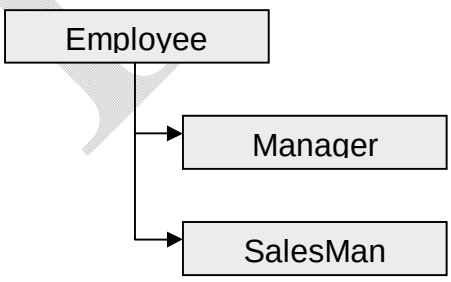
```
    public Student(string n, string usn)
    {
        name=n;
        USN=usn;
    }

    public Student(string n, string u, int s)      //change sem for a particular student
    {
        name=n;
        USN=u;
        Sem=s;
    }

    public static void Main()
    {
        Student s1=new Student("Ramu", "MCA01");      //sem is 5 by default
        Student s2=new Student("Shyam", "CS02", 3);   //sem is set to 3
        --------------
    }
}
```

In the above example, if the field *Sem* would have been specified with ***const*** modifier, then trying to change the value through constructor will generate error. But, ***readonly*** keyword will allow the programmer to keep some fields as constant unless and otherwise specified through constructor.

4.6 The Second Pillar: C#'s Inheritance Supports

Inheritance facilitates code reuse. The inheritance can be either ***classical*** **inheritance** (***is-a*** relationship) or **containment/delegation model** (***has-a*** relationship).

When we establish **is-a** relationship between classes, we are building a dependency between types. The basic idea of classical inheritance is that new classes may influence and extend the functionality of other classes. The hierarchy may look something like –



Now we can say that, Manager is-a Employee and SalesMan is-a Employee.

In classical inheritance, the base classes are used to define general characteristics that are common to all derived classes. The derived classes extend this general functionality while adding more specific behaviors to the class.

Consider an example –

```
public class Emp
{
        protected string Name;
        protected int EmpID;
        ..............
}

public class SalesMan:Emp      //SalesMan is inherited from Emp
{
        int number_of_sales;      // also includes members Name, EmpID
        ...........
}

class Test
{
         public static void Main()
         {
                Emp e=new Emp();
                SalesMan s=new SalesMan();
                ------------
         }
}
```

Controlling Base-Class Creation

Usually, the base class constructors are used to initialize the members of base class. When derived class has few more members and if we need to initialize, then we will write constructor for derived class. In such a situation, there is a repetition of code. For example –

```
        public class Emp
        {
                protected string Name;
                protected int EmpID;

                public Emp(string n, int eid, float s)
                {
                        Name=n;
                        EmpId=eid;
```

```
        }
    }

    public class SalesMan:Emp
    {
        int number_of_sales;

        public SalesMan(string n, int eid, int s)
        {
            Name=n;                 //code repeated
            EmpId=eid;              //code repeated
            number_of_sales=s;
        }
    }
```

If there is more number of data fields in the base class, then in each of the derived class constructors, we need to repeat such assignment statements. To avoid this, C# provides an option to ***explicitly*** call base class constructors as shown below –

```
public class Emp
{
    protected string Name;
    protected int EmpID;

    public Emp(string n, int eid)
    {
        Name=n;
        EmpId=eid;
    }
}

public class SalesMan:Emp
{
    int bonus;
    public SalesMan(string n, int eid, int b): base(n, eid)
    {
        bonus=b;    //other members are passed to base class to initialize
    }
}

class Test
{
```

```
        public static void Main()
        {
                Emp e1=new Emp("Ram", 25);
                SalesMan s1= new SalesMan("Sham", 12, 10);
            ------------
        }
}
```

Multiple Base Classes

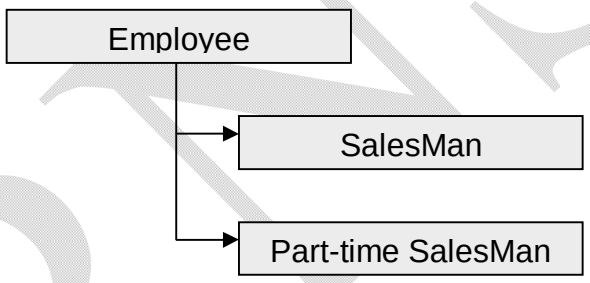
Deriving a class from more than one base class is not possible in C#. That is, multiple inheritance is not supported by C#. When we need to use the properties of more than one class in a derived class, we need to write ***interfaces*** rather than classes. Then we can ***implement*** any number of interfaces to achieve code-reusability.

4.7 Keeping Family Secrets: The *protected* Keyword

The *private* members of a class can not be available to derived classes. So, we will go for ***protected*** keeping the privacy of the data. The *protected* members of a class can not be accessed from outside, but still be available to derived classes.

Preventing Inheritance: *Sealed Classes*

In some of the situations, we may don't want our class to be inherited by any other class. For example –



Here, a class called *Part-time SalesMan* is derived from *SalesMan* class which in-turn is derived from *Employee* class as shown in the diagram. Now, we don't want any more classes to be derived from *Part-time SalesMan* class. To prevent such inheritance, C# provides a keyword ***sealed***. The usage is depicted here-under:

```
public sealed class Part_TimeSalesMan: SalesMan
{
        //body of Part_TimeSalesMan class
}
```

Now any attempt made to derive a class from *Part_TimeSalesMan* class will generate an error:

```
public class Test: Part_TimeSalesMan          //compile-time error!!!
{
        ..............
}
```

Many of the inbuilt classes of C# are sealed classes. One such example is System.String.

Programming for Containment/Delegation

Till now we have discussed *is-a* relationship. Now we will see, how to write program for *has-a* relationship. Consider a class *Radio* –

```
class Radio
{
        public void TurnOn(bool on)
        {
                if(on)
                        Console.WriteLine("Radio is on");
                else
                        Console.WriteLine("Radio is off");
        }
}
```

Now, consider a class *Car* –

```
class Car
{
        int CurrSpeed, MaxSpeed;
        string name;
        bool carIsDead=false;

        public Car() { MaxSpeed=100}
        public Car(string n, int m, int c)
        {
                name=n;
                MaxSpeed=m;
                CurrSpeed=c;
        }
        public void SpeedUp(int a)
        {
                if(carIsDead)
                        Console.WriteLine("Out of order");
                else
```

```
            {
                CurrSpeed+=a;
            }
        }
    }
```

Now, we have two classes viz. *Radio* and *Car*. But, we can not say, *"Car is a Radio"* or *"Radio is a Car"*. Rather, we can say, *"Car has a Radio"*. Now, the *Car* is called ***containing*** Class and *Radio* is called ***contained*** class.

To make the *contained* class to work, we need to re-define *Car* class as –

```
class Car
{       ..........
        private Radio rd=new Radio();
}
```

To expose the functionality of the inner class to the outside world requires ***delegation***. *Delegation* is the act of adding members to the containing class that make use of the functionality of contained class.

```
class Car
{       ..........
        public void Tune(bool s)
        {
            rd.TurnOn(s);       //delegate request to inner object
        }
}
```

To make this function work, in the Main() function we can write –

```
Car c=new Car();
c.Tune(false);
```

Thus by making one class to be a member of other class, we can establish *has-a* relationship. But it is always the job of outer class to define a member function which activates the inner class objects.

4.8 Nested Type Definitions

In C#, it is possible to define a type (enum, class, interface, struct, delegate) directly within the scope of a class. When a type is nested within a class, it is considered as a normal member of that class. For example,

```
class C1
{
        ……..     //members of outer class
        class C2
        {
                …….        //members of inner class
        }
}
```

Nested type seems like *has-a* relationship, but it is not. In containment/delegation, the object of contained class is created within containing class. But, it is possible to create the objects wherever we wish. But in nested types, the inner class definition is within the body of out class and hence, inner class objects can not be created outside the outer class.

4.9 The Third Pillar: C#'s Polymorphic Support

Polymorphism answers the question "how to make related objects respond differently to the same request?" Consider the following example to illustrate the need of polymorphism.

Assume we have a base class *Employee* and two derived classes *Manager* and *SalesMan*.

```
class Employee
{
        …….
        public void Bonus(float b)
        {
                basicSal+=b;
        }
}
class Manager:Employee
{
        ……..
}
class SalesMan: Employee
{
        ……..
}
```

Now, the classes *Manager* and *SalesMan* both contains the method *Bonus(float)*. Thus in the Main() function, we can call *Bonus()* through the objects of *Manger* and *SalesMan* –

```
Manager m=new Manager();
m.Bonus(500);
```

```
SalesMan s=new SalesMan()
s.Bonus(300);
```

Obviously, the *Bonus()* method works same for both the objects. But in reality, the bonus for a SalesMan should be calculated based on number of sales. Manager can be given bonus based on his other performance like successful completion and delivery of a project etc. This means, we need a *Bonus()* method which works differently in two derived classes.

The polymorphic technique of C# provides solution for this problem. With the help of ***virtual*** and ***override*** keywords, we can make same function to behave differently in base-class and all the derived classes. For example,

```
class Employee
{
        ------------------
        public virtual void Bonus(float b)
        {
                basicSal+=b;
        }
}

class SalesMan:Employee
{
        ------------------
        public override void Bonus(float b)
        {
                int salesBonus=0;

                if(numOfSales<=100)
                        salesBonus=10;
                elseif (numOfSales<=200)
                        salesBonus=20;

                base.Bonus(b*salesBonus);      //making use of base class method
        }
}

class Test
{
        public static void Main()
        {
                Employee e=new Employee();
                e.Bonus(100);                  //base class Bonus() is called
```

```
            SalesMan s=new SalesMan()
            s.Bonus(300);                    //derived class Bonus() is called
        }
    }
```

We can see that when the *Bonus()* method is invoked through base class object, the corresponding method will be called. Whereas, when *Bonus()* is invoked with the object *s* of derived class, the ***overridden*** method *Bonus()* will be called.

NOTE that any overridden method is free to call its corresponding base class method using the keyword ***base***. Thus, the overridden method can have its own statements and also can make use of the behaviors of base class virtual method also.

Defining Abstract Classes

Sometimes, the creation of base class objects will not be of any use. For example, an Employee is always identified with a designation. Thus, an employee must be either a Manager or a SalesMan or working at some such other designation. So, creation of the object of class *Employee* is useless. In such situations, we can prevent base class instantiation by making it as ***abstract*.**

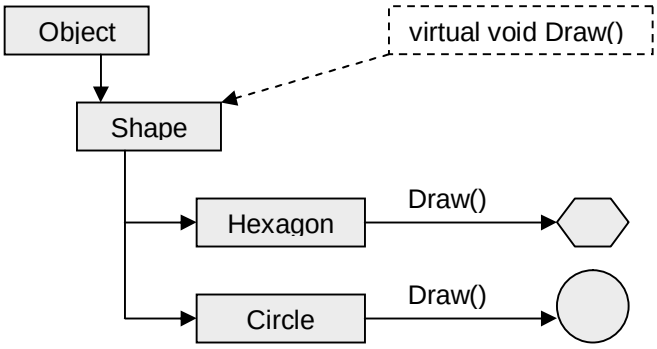
```
public abstract class Employee
{
        ……
}

Employee e=new Employee        //error !!
```

Thus, abstract class is a class which just declares members (data and method) and no object can be created for this class. The member methods of abstract class may be used by the objects of derived class either directly or by overriding them in the derived class.

Enforcing Polymorphic Activity: *Abstract Methods*

An abstract class may contain the definition for methods it is defining. The derived classes are free to use such methods directly or they can override and then use. But in some situations, the definitions of abstract base-class methods may not be useful. The derived classes only should define the required behavior. In this case, we need to **force** the derived classes to **override** the base class methods. This is achieved using ***abstract methods.***

An abstract class can define any number of abstract methods, which will not supply any default implementation. Abstract method is equivalent to ***pure virtual functions*** of C++. The abstract methods can be used whenever we wish to define a method that *does not* supply a *default implementation.* To understand the need for abstract methods, consider the following situation:



Here, each derived class like *Hexagon* and *Circle* of *Shape* has to override *Draw()* method to indicate the methodology of drawing itself. If any derived class forgets to override *Draw()* method, the code may look like –

```
abstract class Shape
{
        public virtual void Draw()
        {
                Console.WriteLine("Shape.Draw()");
        }
}
public class Circle:Shape()
{
        public Circle();
        -------------
}

public class Hexagon:Shape()
{
        ..............
        public override void Draw()
        {
                Console.WriteLine("Hexagon.Draw()");
        }
}

public class Test
{
        public static void Main()
        {
                Circle c= new Circle();
                c.Draw();               //Draw() method of Shape class is called
```

```
            Hexagon h=new Hexagon();
            h.Draw();          //Draw() method of Hexagon class is called
        }
    }
```

In this example, the *Circle* class is not overriding the *Draw()* method. So, when the *Draw()* method is invoked through the object of *Circle* class, the *Draw()* method of *Shape* class itself will be called. Obviously, the *Draw()* method of *Shape* class will not be having any information about how to draw a circle and hence it will be meaningless. This shows that ***virtual methods of base class need not be overridden by derived classes.***

When programmer must be forced to override *Draw()* method, the method of base class must be made ***abstract*** –

```
abstract class Shape
{
    // completely abstract method. Note the semicolon at the end
    public abstract void Draw();
    ----------
}

public class Circle:Shape()
{   -----------------
    public override void Draw()          //has to override
    {
        Console.WriteLine("Circle.Draw()");
    }
}

public class Hexagon:Shape()
{   ..............
    public override void Draw()          //has to override
    {
        Console.WriteLine("Hexagon.Draw()");
    }
}

public class Test
{
    public static void Main()
    {
        Circle c= new Circle();
```

```
            c.Draw();          //Draw() method of Circle class is called

            Hexagon h=new Hexagon();
            h.Draw();          //Draw() method of Hexagon class is called
        }
    }
```

Thus, by making a base class method as *abstract*, we are making use of ***run-time polymorphism or late binding.*** That is, the binding between the object and the method to be invoked is decided only at runtime. This concept is more clear when we re-design our Main() function in the following format:

```
    public static void Main()
    {
        Shape[ ] s={new Circle(), new Hexagon()};

        for(int i=0;i<s.Length;i++)
            s[i].Draw();
    }
```

This may seems to be quite interesting. Till now, we have heard that we can not create objects for abstract classes. But in the above code snippet, we are creating array of objects of base-classes. How? This is very obvious as array of objects stores references to variables but not the objects directly. Thus, in the above example, *s[0]* stores reference to the object of type *Circle* and *s[1]* stores reference to the object of type *Hexagon*.

In the *for* loop, when we use the statement –

```
s[i].Draw();
```

the content (i.e. whether reference to *Circle* or reference to *Hexagon*) of *s[i]* is considered to decide which *Draw()* method to be invoked and the *type* (i.e. here type is *Shape*) of *s[i]* is ignored. This is nothing but run-time polymorphism or late binding.

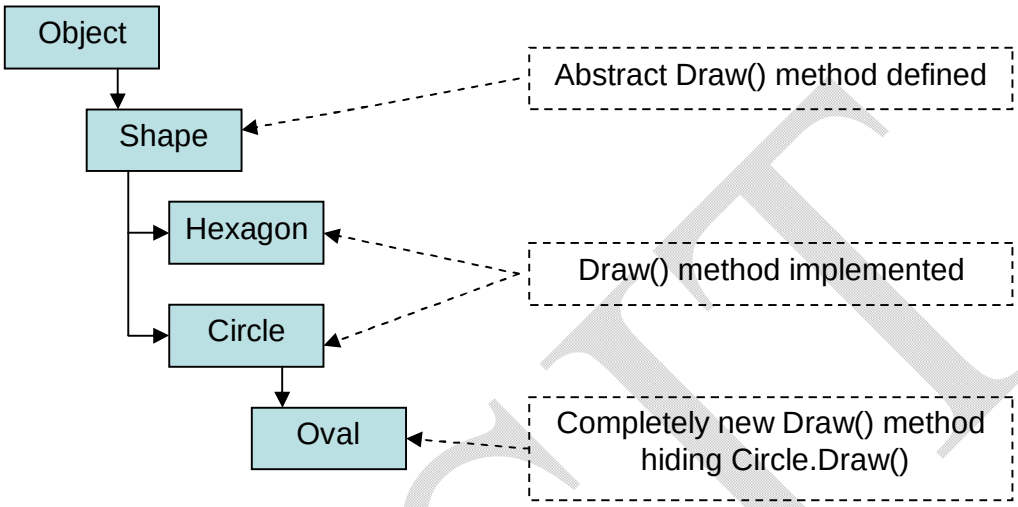
NOTE (Very important):

After dealing with entire story about abstract classes and abstract methods, somebody may feel what exactly is the difference between abstract classes and interfaces? Because just-like abstract classes, interfaces also provide just a prototype of the methods and the implementing classes have to define those methods. (more about interfaces in Chapter 6)

The major difference comes in case of multiple inheritance. We know that we can not derive a class from more than one base-class in C#. That is multiple inheritance is not possible. But, we may have a situation where there is more than one base class and we need to derive a class from all those base classes and then override the virtual methods present in all the base classes. In such a situation, instead of writing abstract base classes, we should write interfaces. Because, a class can be derived from one base class and it can implement many number of interfaces.

Versioning class members

C# provides facility of method hiding, which is logical opposite of method overriding. Assume the following situation:



The class *Circle* is derived from *Shape*. The *Shape* class has an abstract method *Draw()* which is overridden within *Circle* class to indicate how to draw a circle. The class *Oval* behaves similar to *Circle*. But, methodology for drawing an oval is different from that of circle. So, we need to **prevent the *Oval* class from inheriting *Draw()* method.** This technique is known as ***versioning*** a class. This is achieved by using the key word ***new*** as shown below –

```
public class Oval: Circle
{
        public Oval()
        {
                ------
        }

        public new void Draw()
        {
                //Oval specific drawing algorithm
        }
}
```

Now, when we create an object of *Oval* class and invoke *Draw()* method, the most recently used definition of Draw() is called. That is,

```
Oval ob=new Oval();
ob.Draw();          //calls the Draw() method of Oval class
```

Thus, the keyword *new* breaks the relationship between the abstract Draw() method defined by the base class and the Draw() method in the derived class.

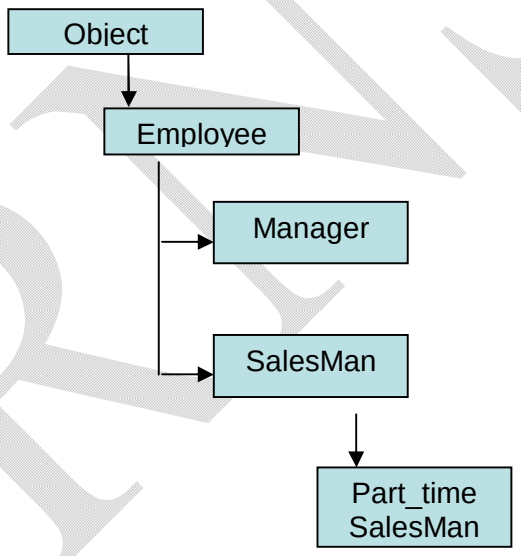
On the other hand, if we wish, we can make use of the base class method itself by using *explicit* casting:

```
Oval obj=new Oval();
((Circle)obj).Draw();          //calls Draw() method of Circle class
```

NOTE:
The method hiding is useful when we are making use of the types defined in another .NET assembly and we are not sure whether that type contains any method with the same name as we are using. That is, in our application we may be using inbuilt classes (with the help of keyword *using*). We are going to create a method *Draw()* in our application, but we are not sure whether any of the inbuilt class contains a method with same name. In such a situation, we can just put a keyword *new* while declaring our method. This will prevent the possible call to built-in class method.

4.10 Casting Between

Till now, we have discussed number of class hierarchies. Now we should know the laws of casting between class types. Consider the following situation, where we have *is-a* relationship –



Keeping the above class hierarchy in mind, the following declarations are purely valid.

```
object ob=new Manager();
Employee e=new Manager();
SalesMan s= new Part_timeSalesMan();
```

In all these situations, the ***implicit casting*** from derived class to base class is done by the C# CLR. On the other hand, when we need to convert base class reference to be stored in derived class object, we should make ***explicit casting*** as –

```
object ob;
Manager m=(Manager)ob;
```

Here, the explicit cast from base class to derived class is done.

Determining the *Type-of* Objects

C# provides three ways to determine if a base class reference is actually referring to derived types:

- Explicit casting
- ***is*** keyword
- ***as*** keyword

We have seen explicit casting just now. The usage of other two methodology is depicted with the help of examples:

```
Object ob=new Manager();
if(ob is Manager)                    //checking whether ob is Manager
        //do something

Employee e;
SalesMan s= e as SalesMan;      //converting Employee type to SalesMan type
//do something with s
```

Numerical Casts

In addition to casting between objects, the numerical conversions also follow similar rules. When we are trying to cast larger (in size of type) type to smaller type, we need to make explicit casting:

```
int x=25000;
byte b=(byte)x;      //loss of information
```

While converting from larger type to smaller type, there is a chance of data-loss.

When we need to cast smaller type to larger type, the implicit casting is done automatically:

```
byte b=23;
int x=b;              //implicit casting
```

There will not be any loss of data during such casting.

Frequently Asked Questions:

1. Explain “is-a” and “has-a” relationship with respect to inheritance. (5)
2. What is encapsulation? What are two ways of enforcing encapsulation? Give examples for both the methods. (10)
3. How do you prevent inheritance using sealed classes? Explain with an example. (5)
4. With an example, explain the concept of abstract classes. (5)
5. What are the basic components of Object oriented programming and how they are implemented? Give examples. (10)
6. Explain abstract methods with example. (5)
7. What do you mean by versioning members? Explain. (5)
8. How do you identify type of an object? Explain. (6)

UNIT 5. Exceptions and Object Lifetime

In this chapter, we are going to study how to handle runtime anomalies using exception handling. Then we will explore how CLR manages object lifetime.

5.1 Meaning of Errors, Bugs and Exceptions

Mistakes are bound to happen during programming. Problems may occur due to bad code like overflow of array index, invalid input etc. So, the programmer needs to handle all possible types of problems. There are three terminologies to define mistakes that may occur.

- ***Bugs:***
  - Errors done by the programmer.
  - Example: making use of NULL pointer, referring array index out of bound, not deleting allocated memory etc.
- ***Errors:***
  - Caused by user of the application.
  - For example: entering un-expected value in a textbox, say USN.
- ***Exceptions:***
  - Runtime anomalies those are difficult to prevent.
  - Example: trying to open a corrupted file, trying to connect non-existing database etc.

5.2 The Role of .NET Exception Handling

The .NET platform provides a standard technique to send and trap runtime errors: structured exception handling (SEH). The beauty of this approach is that developers now have a unified approach to error handling, which is common to all languages targeting the .NET universe. Another bonus of .NET exceptions is the fact that 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, as well as a detailed snapshot of the call stack that triggered the exception in the first place. Furthermore, you are able to provide the end user with help link information that points the user to a URL that provides detailed information regarding the error at hand as well as custom user-defined data.

The Atoms of .NET Exception Handling

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

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

The C# programming language offers four keywords (try, catch, throw, and finally) that allow you to throw and handle exceptions. The type that represents the problem at hand is a class derived from System.Exception.

5.3 The System.Exception Base Class

All user- and system-defined exceptions ultimately derive from the System.Exception base class (which in turn derives from System.Object). Core Members of the System.Exception Type are shown below:

System.Exception **Property**	**Meaning**
HelpLink	This property returns a URL to a help file describing the error in full detail.
InnerException	Used to obtain information about the previous exceptions that caused the current exception to occur. The previous exceptions are recorded by passing them into the constructor of the most current exception. This is a read-only property.
Message	Returns the textual description of a given error. The error message itself is set as a constructor parameter. This is a read-only property.
Source	Returns the name of the assembly that threw the exception.
StackTrace	Contains a string that identifies the sequence of calls that triggered the exception. This is a read-only property.
TargetSite	Returns a Method-Base type, which describes numerous details about the method that threw the exception (ToString() will identify the method by name). This is a read-only property.

5.4 Throwing a Generic Exception

During the program, if any exception occurs, we can throw it to either a specific exception like *FileNotFoundException, ArrayIndexOutOfBoundException, DivideByZeroException* etc. or we can through a generic exception directly using *Exception* class. The object of *Exception* class can handle any type of exception, as it is a base class for all type of exceptions. Here is an example to show how to throw an exception:

```
using System;

class Test
{
        int Max=100;

        public void Fun(int d)
        {
                if(d>Max)
                        throw new Exception("crossed limit!!!");
                else
```

```
                    Console.WriteLine("d={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
d=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 are throwing an exception. We are passing message "*crossed limit"* to a *Message property* of *Exception* class.

Deciding exactly what constitutes throwing an exception and when to throw an exception is up to the programmer.

5.5 Catching Exceptions

When a block of code is bound to throw an exception, it needs to be caught using *catch* statement. Once we catch an exception, we can invoke the members of System.Exception class. Using these members in the *catch* block, we may display a message about the exception, store the information about the error in a file, send a mail to administrator etc. For example,

```
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("{0}", e.Message);
                }
               Console.WriteLine("d={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
d=12

Enter a number: 123
crossed limit!!!
d=123
```

In the above example, we are throwing an exception if *d>100* and is caught. Throwing an exception using the statement –

throw new Exception ("Crossed Limit");

means, creating an object of *Exception* class and passing that object to *catch* block. While passing an object, we are passing the message also, which will be an input for the *Message* property of *Exception* class.

A try block is a section of code used to 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. If the code within try block does not trigger any exception, then the catch block is skipped fully and program will continue to execute further.

Once an exception has been handled, the application will continue its execution from very next point after catch block. In some situations, a given exception may be critical and it may warrant the termination of the application. However, in most of the cases, the logic within the exception handler will ensure the application to be continued in very normal way.

The *TargetSite* Property

- The System.Exception.TargetSite property allows 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.
- However, TargetSite does not simply return a string, but a strongly typed System.Reflection.MethodBase object.
- This type can be used to gather numerous details regarding the offending method as well as the class that defines the offending method.

The *StackTrace* Property

- The System.Exception.StackTrace property allows you to identify the series of calls that resulted in the exception.
- Be aware that you never set the value of StackTrace as it is established automatically at the time the exception is created.
- The string returned from StackTrace documents the sequence of calls that resulted in the throwing of this exception.

```
using System;
class Test
{
        int Max=100;
        public void Fun(int d)
        {
                try
                {
                        if(d>Max)
                                throw new Exception(string.Format("crossed limit!!!"));
                }
                catch(Exception e)
                {
                        Console.WriteLine("{0}",e.Message);                     // crossed limit!!!
                        Console.WriteLine("{0}", e.TargetSite);          //Void Fun(Int32)
                        Console.WriteLine("{0}", e.TargetSite.DeclaringType);   //Test
                        Console.WriteLine("{0}", e.TargetSite.MemberType);              //Method
                        Console.WriteLine("Stack:{0}", e.StackTrace);           //at
Test.Fun(Int32 d)
                }
                Console.WriteLine("d={0}", d);
          }
          public static void Main()
          {
```

```
                Test ob=new Test();
                Console.WriteLine("Enter a number:");
                 int d=int.Parse(Console.ReadLine());          //assume d is entered as 123
                ob.Fun(d);
            }
}
```

The *Helplink* Property

- The HelpLink property can be set to point the user to a specific URL or standard Windows help file that contains more detailed information.
- By default, the value managed by the HelpLink property is an empty string.
- If you wish to fill this property with an interesting value, you will need to do so before throwing the System.Exception type.

```
using System;
class Test
{
        int Max=100;
        public void Fun(int d)
        {
                try
                {
                        if(d>Max)
                        {
                                Exception ex= new Exception("crossed limit!!!");
                                ex.HelpLink="g:\\Help.doc";
                                throw ex;
                        }
                }
                catch(Exception e)
                {
                        Console.WriteLine("{0}", e.HelpLink);                    // G:\Help.doc
                }
                Console.WriteLine("d={0}", d);
        }
        public static void Main()
        {
                Test ob=new Test();
                Console.WriteLine("Enter a number:");
                 int d=int.Parse(Console.ReadLine());                  //say d=345
                 ob.Fun(d);
        }
}
```

5.6 CLR System Level Exceptions

The .NET base class libraries define many classes derived from System.Exception. For example, the System namespace defines core error objects such as ArgumentOutOfRangeException, IndexOutOfRange-Exception, StackOverflowException etc. Other namespaces define exceptions that reflect the behavior of that namespace. For ex. System.Drawing.Printing defines printing exceptions, System.IO defines IO-based exceptions, System.Data defines database-centric exceptions, and so on.

Exceptions that are thrown by the methods in the BCL are called *system exceptions*. These exceptions are regarded as non-recoverable, fatal errors. System exceptions derive directly from a base class named System.SystemException, which in turn derives from System.Exception (which derives from System.Object):

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

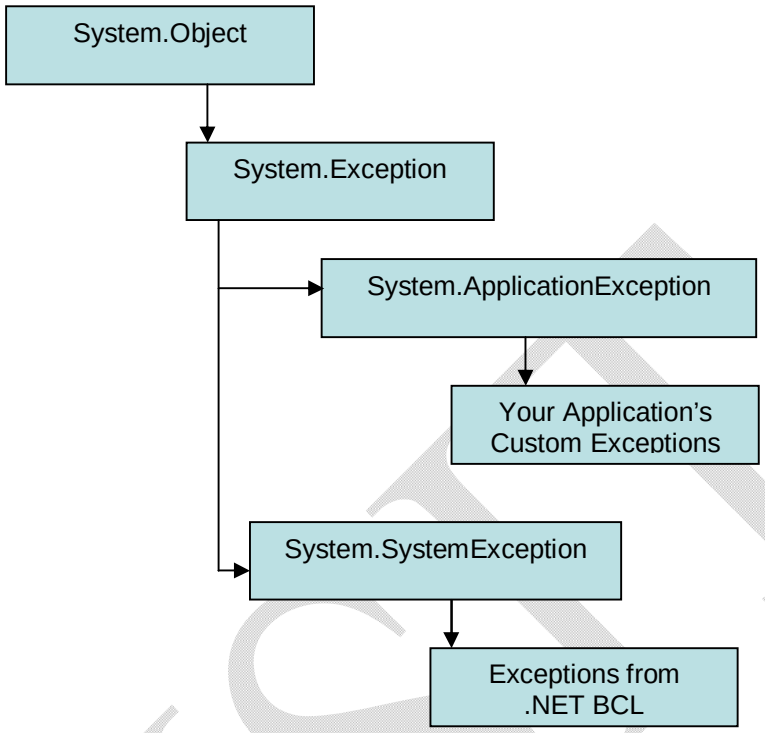
The System.SystemException type does not add any additional functionality beyond a set of constructors.

5.7 Custom Application Level Exceptions

All .NET exceptions are class types and hence we can create our own application-specific exceptions. Since the System.SystemException base class represents exceptions thrown from the CLR, we may naturally assume that we should derive your custom exceptions from the System.Exception type. But, the best practice is to derive from the System.ApplicationException type:

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

The purpose of System.ApplicationException is to identify the source of the (nonfatal) error. When we handle an exception deriving from System.ApplicationException, we can assume that exception was raised by the code-base of the executing application rather than by the .NET BCL. The relationship between exception-centric base classes are shown –



Truly speaking, it is not necessary to derive a custom exception from System.ApplicationException class, rather it can directly be derived from more generic System.Exception class. To understand various possibilities, now we will build custom 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 *strongly typed exception* that represents the unique details of our current problem. The first approach is defining a new class derived directly from System.Exception. 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 Ikon”);
                }
                catch(CarIsDeadException e)
                {
                        Console.WriteLine(“Method:{0}”, e.TargetSite);
                        Console.WriteLine(“Message:{0}”, e. Message);
                }
        }
        ……….
}
```

Note that, the custom exceptions are needed only when the error is tightly bound to the user-defined class issuing the error. For example, a File class may throw number of file-related errors; Student class may throw errors related to student information and so on. By writing the custom exception, we provide the facility of handling numerous exceptions on a name-by-name basis.

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

We have discussed that; exceptions can be system-level or application-level. If we want to clearly mark our exception to be thrown by the application itself and not by any chance by BCL, we can redefine as –

```
public class CarIsDeadException: ApplicationException
{
        //body
}
```

5.8 Handling Multiple Exceptions

We have seen a situation where a try block will have corresponding catch block. But in reality, we may face a situation where the code within try block may trigger multiple possible exceptions. For example,

```
public void SpeedUp(int delta)
{
        try
        {
                if(delta<0)
                        throw new ArgumentOutOfRangeException("Ford Ikon");

                speed=current_speed + delta;
                if(speed>max_speed)
                        throw new CarIsDeadException("Ford Ikon");
        }
        catch(CarIsDeadException e)
        {
                Console.WriteLine("Method:{0}", e.TargetSite);
                Console.WriteLine("Message:{0}", e. Message);
        }
```

```
        catch(ArgumentOutOfRangeException e)
        {
                Console.WriteLine("Method:{0}", e.TargetSite);
                Console.WriteLine("Message:{0}", e. Message);
        }
}
```

In the above example, if *delta* is less than zero, then *ArgumentOutOfRangeException* is triggered. If the speed exceeds *MaxSpeed*, then *CarIsDeadException* is thrown. While constructing multiple catch blocks for a single try block, we must be aware that it will be processed by the *nearest available* catch block. For example, the following code snippet generates compile-time error!!

```
try
{
        ----------
}
catch(Exception e)
{
        ----------
}
catch(CarIsDeadException e)
{
        ----------
}
catch(ArgumentOutOfRangeException e)
{
        ----------
}
```

Here, the class *Exception* is a base class for all user-defined and built-in exceptions. Hence, it can handle any type of exceptions. Thus, if *Exception* is the first catch-block, the control will jump to that block itself and the other two exceptions are *unreachable.* Thus, during multiple exceptions, we have to design very structured format. The very first catch block should contain very specific exception or the most derived type in inheritance hierarchy. Whereas the last catch block should be the most general or the 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.

Re-throwing Exceptions
It is possible to re-throw an error to the previous caller. For example–

```
try
{
        ----------
}
catch(CarIsDeadException e)
{
        //handle CarIsDeadException partially or do something
        throw e;
}
```

In the above code, the statement *throw e* will throw the exception again to *CarIsDeadException.*

5.9 The *Finally* Block

The try/catch block may be added with an optional *Finally* block. The *finally* block contains a code that always executes irrespective of any number of exceptions that may interfere with the normal flow of execution. For example, we may want to turn-off radio within a car before stopping the car due to any reason. Programmatically speaking, before exiting from Main() function, we may need to turn-off radio of a car. Then we can write –

```
public static void Main()
{
        try
        {
                //increase speed
        }
        catch(CarIsDeadException e)
        {
                ----------
        }
        catch(ArgumentOutOfRangeException e)
        {
                ----------
        }
        finally
        {
```

```
                CarName.TurnOffRadio();
        }
}
```

NOTE:

- The finally block is useful in following situations –
  When we want to
  - clean-up any allocated memory
  - Close a file which is in use
  - Close a database connection to a data source
- The finally block will be executed even if our try block does not trigger any exception.
- This is especially helpful if a given exception requires the termination of the current application.

5.10 Dynamically Identifying Application and System Level Exceptions

In the previous examples, we have handled each exception by its specific class name. Iternatively, if we want to generalize our catch blocks in a 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.

5.11 Understanding Object Lifetime

By the example programs we have seen in previous four chapters, we can note that we have never directly de-allocated an object from memory. That is there is no ***delete*** keyword in C#. Rather, .NET objects are allocated onto a region of memory termed as ***managed heap***, where they will be automatically de-allocated by the CLR at *some time later.*

Thus, the golden rule of .NET memory management is –
Allocate an object onto the managed heap using the new keyword and forget about it.

Once an object is created using *new*, the CLR removes it when it is no longer needed. That is, the CLR removes an object from the heap when it is unreachable by the current application. For example,

```
public static void Main()
{
        Car c=new Car("Ford Ikon");
        ..................
}
```

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 can not 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.

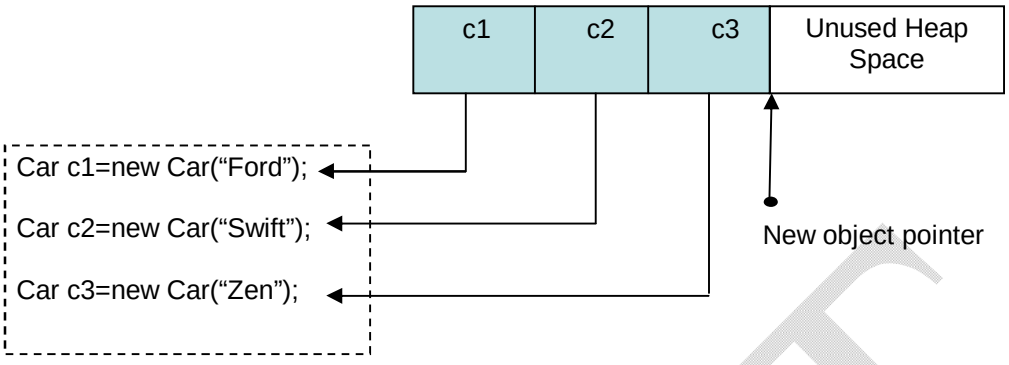
5.12 The CIL of *new*

Here we will discuss what actually happens when *new* keyword is used.

- When C# encounters the *new* keyword, it will produce a CIL ***newobj*** instruction to the code module.
- Note that, the managed heap is not just a raw portion of memory accessed by the CLR.
- The .NET garbage collector will compact empty blocks of memory, when needed, for the purpose of optimization.

- To help this process, the managed heap maintains a pointer, referred as ***new object pointer,*** that identifies exactly where the ***next object*** will be placed on the heap.
- After these tasks, the ***newobj*** instruction informs the CLR to perform the following sequence of events:
    - Calculate the total amount of memory required for the object to be allocated. If this object contains other internal objects (i.e. *has-a* relationship and/or nested type member), they are also added-up. And the memory required for each base class is also considered (i.e *is-a* relationship).
    - The CLR then examines the managed heap to ensure that there is enough space for the object to be allocated. If so, the object's constructor is called and a reference to the object in the memory is returned.
    - Finally, before returning the reference, the CLR will move the new object pointer to point to the next available slot on the managed heap.

The entire process is depicted here-under:



5.13 The Basics of Garbage Collection

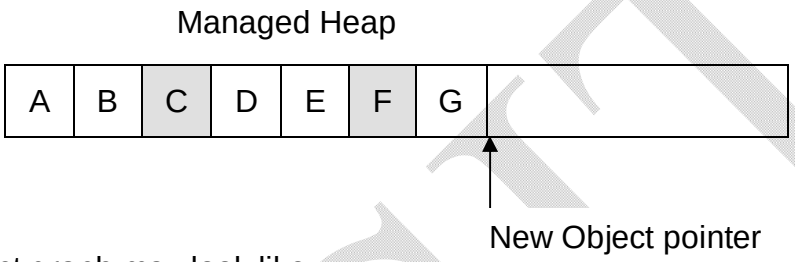
- After creating so many objects, the managed heap may become full.
- When the ***newobj*** instruction is being processed, if the CLR determins that the managed heap does not have sufficient memory to allocate the requested type, it will perform a garbage collection in an attempt to free-up the memory.
- Thus, the next rule of garbage collection is:

> ***If the managed heap does not have sufficient memory to allocate a new object, a garbage collection will occur.***

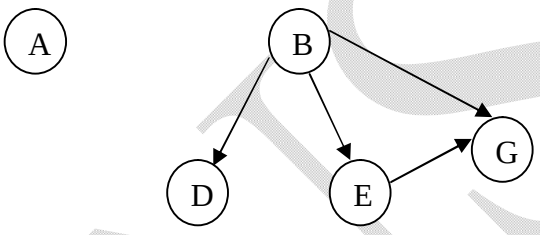
- Now the question arise: how CLR is able to determine an object on the heap that it is no longer needed and destroy it?
- To answer this question, we should know ***application roots.***
- A ***root*** is a storage location containing a reference to an object on the heap.
- In other words, a ***root*** is a variable in our application that points to some area of memory on the managed heap.
- The root can fall into any of the following categories:
  - Reference to global objects (Though global objects are not allowed in C#, raw CIL does permit allocation of global objects)
  - Reference to static objects
  - References to local objects within a given method
  - References to object parameters passed into a method
  - Any CPU register that references a local object
- When a garbage collection occurs, the runtime (CLR) will check all objects on the managed heap to determine if it is still in use (or rooted) in the application.
- To do so, the CLR will build an ***object graph***, which represents each object on the heap that is still reachable.
- The object graph documents all co-dependencies for the current object.
- The CLR will ensure that all related objects are considered before a possible garbage collection through the construction of an object graph.

- The CLR will never graph the same object twice, and thus avoids the circular reference count.

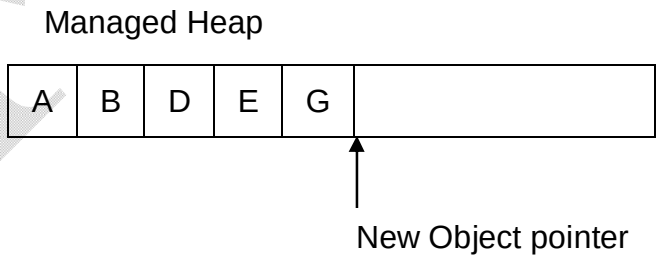
To illustrate this concept, assume that the managed heap contains a set of objects named A, B, C, D, E, F, and G. During a garbage collection, these objects (as well as any internal object references they may contain) are examined for active roots. Once the graph has been constructed, unreachable objects, say, the objects C and F are marked as garbage as shown in the following diagram:



Now, a possible object graph may look like –



Here, the arrows indicate *depends on* or *requires.* For example, E depends on G, B depends on E and also G, A is not depending on anything etc. Once an object has been marked for collection (here, C and F), they are not considered for object graph creation and are swept from the memory. At this point, the remaining space on the heap is compacted, which in turn will cause the CLR to modify the set of active application roots to refer to the correct memory location (this is done automatically and transparently). Then, the new object pointer is readjusted to point to the next available slot. Following diagram depicts it –



5.14 Finalizing a Type

From the previous discussion, we can easily make out that the .NET garbage collection scheme is ***non-deterministic*** in nature. In other words, we can not determine exactly when an object will be de-allocated from the memory. This approach seems to be quite good because, we, the programmers need not worry once the object has been created. But, there is a possibility that the objects are holding ***unmanaged resources*** (Win32 files etc) longer than necessary.

When we build .Net types that interact with unmanaged resources, we like to ensure that this resource is released in-time rather than waiting for .NET garbage collector. To facilitate this, the C# provides an option for overriding the virtual System.Object.Finalize() method. The default implementation of Finalize() method does nothing!! Note that we need to override it only if we are making use of unmanaged resources. Otherwise, the C# garbage collector will do the job.

C# will not allow the programmer to directly override the Finalize() method.

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

Rather, we need to use a C++ -type destructor syntax:

```
public class Test
{
        ~Test()
        {
                ……..
        }
}
```

Indirectly Invoking System.Object.Finalize()

We have discussed till now that the .NET runtime will trigger garbage collector when it requires more memory that what is available at heap. But also note that, the finalization will automatically take place when an ***application domain*** or ***AppDomain*** is unloaded by CLR. Application domain can be assumed to be Application itself. Thus, once our application is about to shut down, the finalize logic is triggered.

Thus, the next rule of garbage collection is:

When an AppDomain is unloaded, the Finalize() method is invoked for all finalizable objects.

For illustration, consider the following example –

```
using System;
```

```
class Test
{
        public Test()
        { }

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

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

Output:

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

We can see that, once the program control goes out the scope of Main() function, the destructor is called for the object *t*. In the destructor or the finalizer, we need to write the code to release the resources that may be held by the object *t.*

5.15 The Finalization Process

Though finalizer seems to be good to use, it is not advised unless the objects in our application are using unmanaged resources. The good programming practice is to avoid Finalize() method, as finalization takes time.

- When an object is placed on a heap using *new*, the CLR automatically determines whether this object supports a user-defined Finalize() method.
- If yes, the object is marked as ***finalizable*** and a pointer to this object is stored on an internal queue names as ***the finalization queue.***
- The 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 checks every entry on the finalization queue and copies the object from the heap to another CLR-managed structure termed as ***finalization reachable table (f-reachable).***

- At this moment, a separate thread is produced to invoke the Finalize() method for each object on the f-reachable table at the *next garbage collection.*
- Thus, when we build a custom-type (user-defined type) that overrides the System.Object.Finalize() method, the .NET runtime will ensure that this member is called when our object is removed from the managed heap.
- But this will consume time and hence affects the performance of our application.

5.16 Building an Ad Hoc Destruction Method

We have seen that the objects holding unmanaged resources can be destroyed using Finalize() method. But the process of finalization is time consuming. C# provides an alternative way to avoid this problem of time consumption.

The programmer can write a custom ad hoc method that can be invoked manually before the object goes out of the scope. This will avoid the object being placed at finalization queue and avoid waiting for garbage collector to clean-up. The user-defined method will take care of cleaning up the unmanaged resources.

```
public class Car
{       …….
        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, the .NET class libraries define an interface named ***IDisposable***. This interface contains a single member ***Dispose():***

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

Now, our application can implement this interface and define *Dispose()* method. Then, this method can be called manually to release unmanaged resources. Thus, we can avoid the problems with finalization.

```
public class Car: IDisposable
{       ……..
        public void Dispose()
        {
                //code to clean up unmanaged resources
```

```
        }
}

class Test
{       ……..
        public static void Main()
        {
                Car c=new Car("Ford");
                ……..
                c.Dispose();
                ……..

        }       //c still remains on the heap and may be collected by GC now
}
```

Thus, another rule for working with garbage collection is:

Always call Dispose() for any object in the heap. The assumption is : if there is a Dispose() method, the object has some clean up to perform.

Note that, for a single class, it is possible to have C# - style destructor and also to implement IDisposable interface for defining Dispose() method.

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.

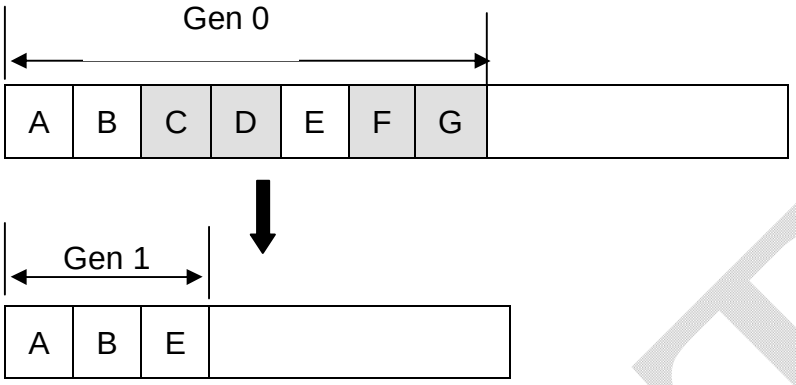
5.17 Garbage Collection Optimizations

Till now we have seen two methodologies for cleaning user-defined types. Now, we will discuss little deeply about the functionality of .NET garbage collector.

- When CLR is attempting to locate unreachable objects, it does not literally search through every object placed on the managed heap looking for orphaned roots.
- Because, doing so will consume more time for larger applications.
- To optimize the collection process, each object on the heap is assigned to a given ***generation***.
- The idea behind generation is as follows:
  - If an object is on the heap since long time, it means, the object will continue to exist for more time. For example, application-level objects.
  - Conversely, if an object has been recently placed on the heap, it may be dereferenced by the application quickly. For example, objects within a scope of a method.
- Based on these assumptions, each object belongs to one of the following generations:
  - ***Generation 0:*** Identifies a newly allocated object that has never been marked for collection.
  - ***Generation 1:*** Identifies an object that has survived a garbage collection sweep (i.e. it was marked for collection, but was not removed due to the fact that the heap had enough free space)
  - ***Generation 2:*** Identifies an object that has survived more than on sweep of the garbage collector.

- Now, when a collection occurs, the GC marks and sweeps all generation 0 objects first.
- If required amount of memory is gained, the remaining objects are promoted to the next available generation.

To illustrate how an object's generation affects the collection process, consider the following diagram –



- If all generation 0 objects have been removed from heap and still more memory is necessary, generation 1 objects are checked for their reachability and collected accordingly.
- Surviving generation 1 objects are then promoted to generation 2.
- If the garbage collector still requires additional memory, generation 2 objects are checked for their reachability.
- At this point, if generation 2 objects survive a garbage collection, they remain at that generation only.
- Thus, the newer objects (local variables) are removed quickly and older objects (application level variables) are assumed to be still in use.
- This is how, the GC is able to quickly free heap space using the generation as a baseline.

5.18 The System.GC Type

The programmer can interact with the garbage collector using a base class ***System.GC***. This class provides following members:

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

We have discussed earlier that 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 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.

Interacting with Generations

It is possible to find generation of every object at any moment of time in the program.

```
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));        //C1 is Gen 0
                Console.WriteLine("c2 is Gen {0}", GC.GetGeneration(c2));        //C2 is Gen 0
                Console.WriteLine("c3 is Gen {0}", GC.GetGeneration(c3));        //C3 is Gen 0
                Console.WriteLine("c4 is Gen {0}", GC.GetGeneration(c4));        //C4 is Gen 0

                c1.Dispose();                   //Within Dispose() of One
                c3.Dispose();                   //Within Dispose() of Three

                GC.Collect(0);

                Console.WriteLine("c1 is Gen {0}", GC.GetGeneration(c1));        //C1 is Gen 1
                Console.WriteLine("c2 is Gen {0}", GC.GetGeneration(c2));        //C2 is Gen 1
                Console.WriteLine("c3 is Gen {0}", GC.GetGeneration(c3));        //C3 is Gen 1
                Console.WriteLine("c4 is Gen {0}", GC.GetGeneration(c4));        //C4 is Gen 1
        }
}                                       //Within Destructor of Four
                                        //Within Destructor of Two
```

The output of above program may be a bit surprising. Using statement *GC.Collect(0)*, we are forcing the garbage collection to occur for generation 0 objects. After looking at next output lines, we can see that, all the four objects have been moved to generation 1 instead of getting destroyed. This is because, when we force garbage collection to start, it will realize that enough memory is there in the heap and is not necessary to sweep the objects. Hence, the generation 0 objects are survived from collection and they are moved to generation 1.

Since Dispose() method is called for the objects c1 and c3, the finalization will happen only for c2 and c4 after coming out of Main() method.

Frequently Asked Questions:

1. What do you understand by exception in C#? Illustrate the use of System.Exception base class in throwing generic exceptions. (10)
2. How does .NET framework manage garbage collection? Explace using IDisposable interface. (10)
3. Differentiate between bugs, errors and exceptions. Explain the concepts of .NET exception handling with valid example code. (8)
4. Explain the following properties: TargetSite, StackTrace, HelpLink. (6)
5. What do you mean by custom exception? Write a program to build a custom exception which raises an exception when the argument passed is a negative number. (10)
6. Explain how garbage collection is optimized in .NET? (5)
7. When do you override the virtual System.Object.Finalize() method? How to implement it using destructors? (6)