C# Programming

Study Material for MS-32

Directorate of Distance Education
Guru Jambheshwar University of Science &Technology, Hisar

Study Material Prepared by Manoj Kumar and Arun Chowdhary

Published by:Excel Books, A-45, Naraina, Phase I, New Delhi-110 028

Published by Anurag Jain for Excel Books, A-45, Naraina, Phase I, New Delhi-110 028 and printed by him at Excel Printers, C-206, Naraina, Phase I, New Delhi - 110 028

CONTENTS

UNIT

1

INTRODUCTION TO C#

L E A R N I N G O B J E C T I V E S

After studying this unit, you should be able to:

- Describe common language specification
- Understand about C# programming basics

U N I T S T R U C T U R E

1.1 INTRODUCTION

This chapter will introduce you to C# which Microsoft introduced as an attempt to have a single computer language that has all the features that a modern computer would like it to have. It has several features and capabilities, which enable programmers to overcome the limitations of other programming languages such as Visual Basic, C and C++ and Java. You will also learn about the .NET platform which is the most important component essential to C# programming as the codes compiled by C# program can run only on the .NET platform. You will be introduced to some of the fundamental aspects and programming basics of C.

1.2 OVERVIEW

The needs of a computer programmer and the features they demand from a programming language keep on changing with time. Once C was supposed to be answer to all the questions in programming. Though to some extent it is true, yet the need of simpler programming languages was immediately felt. This very fact has compelled the programming language designers to continuously revise the existing programming languages. The rapid pace of improvements in related technologies demands that programming tools must improve even more rapidly to catch up with them. This is a continuous process and in this process comes a point where a revised version of a language is so much detached from the original counterpart that it takes form of an altogether new language.

Due to ongoing struggle for creating the ultimate programming language we have ended up having so many programming languages today - one catering to a specific need better than the others.

Latest in the sequel of this evolutionary development, Microsoft introduced a new programming language represented symbolically as C# and pronounced see-sharp. In the recent past it was observed that both languages C and C++ were very popular among

programmers due to the programming-power they provided to them. However, Visual Basic was found to be popular because of its simplicity. Some languages have much desirable features while lack in other useful features. In short, no single computer language has all the features that a modern computer would like it to have. C# is an attempt of Microsoft to provide just that.

The designers of C# have lifted good features from the existing programming languages and alleviated them from arcane drudgery by reducing the limitations or removing them completely. Let us have a look at what limitations in existing programming languages are and how C# enables the programmers to overcome them.

- C# has retained the code-simplicity of Visual Basic. VB programmers would not feel the migration- pain while shifting to C#. VB suffered severely from its limited OOP capabilities. C# offers true OOP capabilities to the VB programmers without sacrificing the simplicity of the code.
- C# has a significant resemblance with the syntax of C++ and Java. It has retained the usability of OOP and API that C++ provides but has done away with the much error-prone memory management, much difficult pointer manipulations and incomprehensible code-structures. C++ programmers have long waited for these enhancements.
- Java programmers felt handicapped when they wanted to use the existing codes written in alien languages. Though Java offers platform-independence it is not truly language-independent. A Java application has to be written in Java end-to-end. It has very limited capability to use non-Java API's. C# heralds a good news to Java programmers by providing seamless integration with millions of lines of codes written in non-Java languages.
- Internet programming has always been a difficult enterprise. Because of the availability of great numbers of incompatible and semi-compatible technologies Internet programmers never had a blissful moment. C# does not differentiate between a normal application from Internet application. This very fact is enough a reason for the Internet programmers to migrate to C#.

The most important component that is essential to C# programming is the .NET platform. The codes produced by compilation of a C# program can run only on the .NET platform. However, as you will see in the forthcoming sections, this very aspect is not a limitation as it may sound but a means to achieve total platform-independence.

Microsoft's new release of Visual Studio includes all the necessary tools for C# programming on .NET platform. It also includes VB.NET, ASP.NET apart from other language tools. The studio has been named Visual Studio.NET version 7.0. The studio is an extension to the previous version and hence all the previous versions of tools and libraries are also accessible to the programmers.

1.3 .NET PLATFORM

Dot Net (.NET) is the Microsoft's latest offering for all the programmers belonging to every school of thought. Dot Net has introduced radical changes in the programming paradigm albeit only to make the task simpler. Moreover, it does not take too much effort from any programmer to switch over to this platform.

Soon after the official announcement of the first release, as many as 30 different programming languages have initiated .NETting themselves. No matter whether you program in C, C++, Java, Cobol, Smalltalk, Pascal or any other language, you can always harness the huge advantages that .NET has to offer. The languages, which have reincarnated themselves into .NET version are referred to as .NET aware languages.

.NST is itself not a programming language. It is rather a programming framework that provides all the goodies of all the programming languages under one roof. You can write one module of an application in Java, another in Cobol, pick up a module written in C and integrate them all to make a single application. Could you imagine anything like this before!

.NET platform provides both integrated development environment and a uniform runtime environment. The framework is a new programming platform aptly suited for application development and deployment in highly distributed environment of such as the Internet.

A basic understanding of .NET framework is desirable to start programming in C#. .NET Framework has four main components:

Common Language Runtime (CLR)

Language Runtime or simply runtime is an interface that provides services to a program while executing its code. You probably already would know about Java runtime - Java Virtual Machine (JVM); Visual Basic 6.0 runtime module - msvbvm60.dll etc.

These runtimes provide the necessary linkages to the existing codes that ultimately make the execution possible. It is responsible for loading and managing the code during its execution. CLR is the very core of the .NET Framework. It acts as an agent that manages code at execution time. It provides services such as memory management; thread management, remoting and enforcing strict safety and accuracy of the code. In relation to runtime, codes can be either managed or unmanaged. Code that targets the runtime is known as managed code; code that does not target the runtime is known as unmanaged code.

Since .NET has to support several .Net aware programming languages, it must provide a common runtime that manages codes written in all these languages. CLR does exactly that - it provides a common language runtime environment. It is this common runtime that makes .NET operating system independent.

The .NET Common Language Runtime is implemented as a dynamic link library - mscoree.dll. The programs written for .NET runtime will run on any operating system where .NET runtime is available.

Common Type System (CTS)

Every programming language deals with certain data types. However, the data types are often incompatible between languages. For instance, an integer type data may occupy different number of bytes in the memory in different languages. This inconsistency must be resolved for a common language runtime.

CTS is the specification for such a common data type. It formally specifies the syntax and semantics of how different data types must be declared in the .NET aware programming languages so that CLR may support them while execution. In order that all these languages may share their codes among each other, they must conform to these specifications.

Specifically CTS defines the following common types:

- ***CTS Class Types:*** A class is the core implementation of object-oriented programming paradigm. It encapsulates data and methods and makes inheritance possible. Every .NET aware language must support a class structure. The declaration must also specify whether a class can be sub-classed or not; whether the class implements any interface or not; whether the class is abstract or not and the visibility of the class. All the classes implicitly derive from the base class System.Object.
- ***CTS Structure Types:*** A structure is a data type that comprises of a group of other data types and yet can be treated as a single unit. Structure in .NET world is very similar to structures in other platforms. .NET also allows constructors in a structure which can be used to initialize its members at the time of structure creation.
- ***CTS Interface Types:*** Interfaces are abstract classes. They cannot be instantiated. They simply specify the methods without providing any code for the same. It is up to the classes and sub-interfaces to implement an interface by providing code for it. .NET interfaces can derive from multiple interfaces.
- ***CTS Enumeration Type:*** An enumeration data type allows programmers to group a number of data items in such a way that they can be accessed by their cardinal number. Every enumeration type must derive from System.Enum base class.

- ***CTS Delegate Types:*** If you are aware of language C, you would also probably be aware of function pointers. A function pointer is an address in the memory where a function is stored. A function loaded in the memory may be called by its name or through its function pointer. The problem with C function pointer is that it happens to be just an address and therefore it may not always be safe to use.

  .NET delegate type is loosely equivalent to C function pointer. However, in contrast, it is a class rather than just an address. It must derive from MulticastDelegate base class. For this reason safety of function calling is guaranteed.

- ***CTS Intrinsic Data Types:*** .NET defines clearly and rigorously all the atomic data types. As expected even these intrinsic data types are also classes. Each .NET aware language must conform to these specifications.

1.4 COMMON LANGUAGE SPECIFICATION (CLS)

Every programming language has its own syntactical structure of writing programs. For instance some languages concatenate two strings using + and some using &. While some languages use return reserved word to return a value from a function, some simple use assignment. The respective compilers will flag errors if a program does not adhere to the specific syntactic rules of that language.

.NET provides a common set of rules that all .Net aware languages must conform to. Their compilers have been designed to support these rules. This common set of rules is called CLS.

Class Library (CL)

The libraries accompanied with each language need not be thrown away. .NET provides mechanism through which these libraries can be used in a .NET aware program. Strictly speaking there is no class library in C#. However, it can use any class provides with the .NET framework.

.Net organizes all these binary classes into neatly packed assemblies and access them using namespace. These namespaces are uniformly accessible to all the .NET aware languages. Here is a synopsis of how the System.Console class is accessed by VB.NET, C# and .NET aware C++ (aka Managed C++ or simply MC++).

```
Code in VB.NET

   Imports System

   Public Module Test

Sub Main()

   Console.WriteLine("Welcome to the .NET world")

End Sub

End Module

Code in C#

using System;

public class  Test()

{

   public static void Main()

   {

      Console.WriteLine("Welcome to the .NET world");

   }

}
```

```
Code in MC++

using namespace System;

void main()

{

   Console.WriteLine("Welcome to the .NET world");

}
```

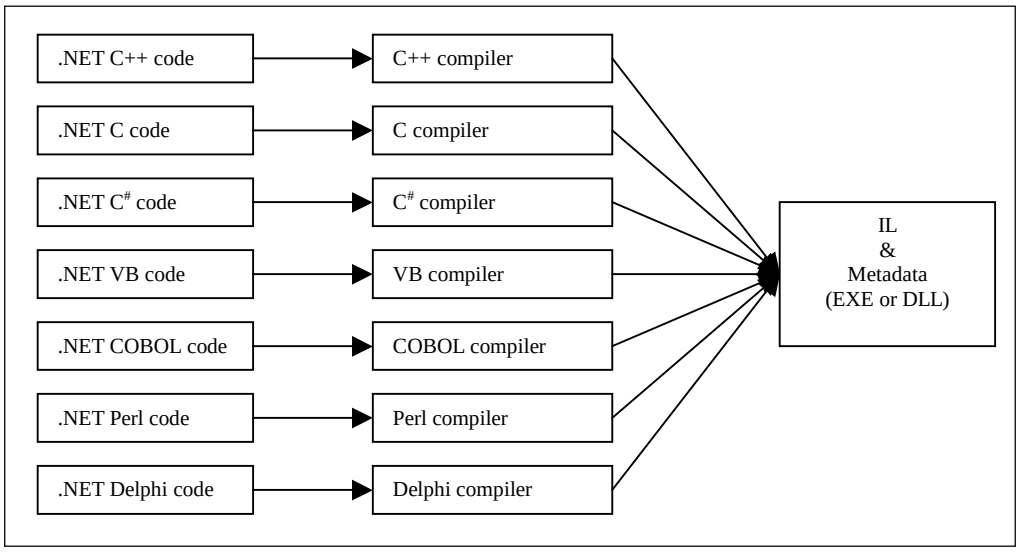
.NET programmers have access to a rich class library organized into namespaces.. A program on .NET platform must use at least System namespace which defines Object root class. In order to use a class, structure, methods, enumeration or any such construct packed into a namespace the program must specify the reference to the namespace. As you have seen in the example codes given above, VB.NET uses Imports clause; C# uses using clause; and MC++ uses using namespace clause.

Student Activity 1

1. What are some of limitations of other programming languages that C# enables programmers to overcome?
2. What is .Net Framework? Explain the architecture of .Net.
3. What are the different functionalities associated with Common Language Runtime (CLR)?
4. What are the common data types specified by the component CTS of .Net framework?
5. What do you mean by a namespace? List down some of the Namespace available in .Net.
6. What are .NET aware languages?

Compiling .NET programs

Compiled .NET programs are not targeted to a specific operating system or a specific machine. Rather than that like Java it compiles the programs for .NET platform in a language called Intermediate Language (or IL). Moreover, no matter which .NET aware language you choose to program with they generate same IL code. This feature makes .NET truly language-independent as is depicted below.



Though the source codes are language specific, the compiled codes are in IL, which is independent of the language used for programming. This IL code can be converted into platform-specific code at the runtime. Since all the languages produce same IL code, these

codes can be integrated into one another making .NET fully language-integrable. This IL code then can be executed on any platform that supports .NET runtime environment.

Visual Studio.NET 7.0 provides a very useful tool that allows one to view the structure as well as the IL code of an executable (exe) or dynamic link library (dll) assembly. This tool is called - Intermediate Language Deassembler. This tool can be invoked by executing ildasm.exe usually located in the Bin folder of Visual Studio.NET 7.0 installation.

To make the point clear take a look at the following source code examples.

VB.NET source code:

```
Module ModuleOne
  Class math_sum
    Public Function SumIt(ByVal x As Integer, ByVal y as Integer)As
    Integer
          SumIt = x + y
    End Function
  End Class
  Sub Main()
    Dim x1 as As Integer
    Dim x2 as New math_sum()
    x1 = x2.SumIt(23, 10)
    Console.WriteLine("23 + 10 = {0}.", x1)
  End Sub
End Module
```

The resulting IL code for the Add method looks like:

```
.method public instance int32 SumIt(int32 x, int32 y) ill managed
{
  //Code size   11 (0xb)
    .maxstack  2
    .locals    init ([0] int32 SumIt)
    IL_0000:   nop
    IL_0001:   ldarg.1
    IL_0002:   ldarg.2
    IL_0003:   add.ovf
    IL_0004:   stloc.0
    IL_0005:   nop
    IL_0006:   br.s IL_0008
    IL_0008:   nop
    IL_0009:   ldloc.0
    IL_000A:   ret
} //end of method ModuleOne$math_sum::SumIt
```

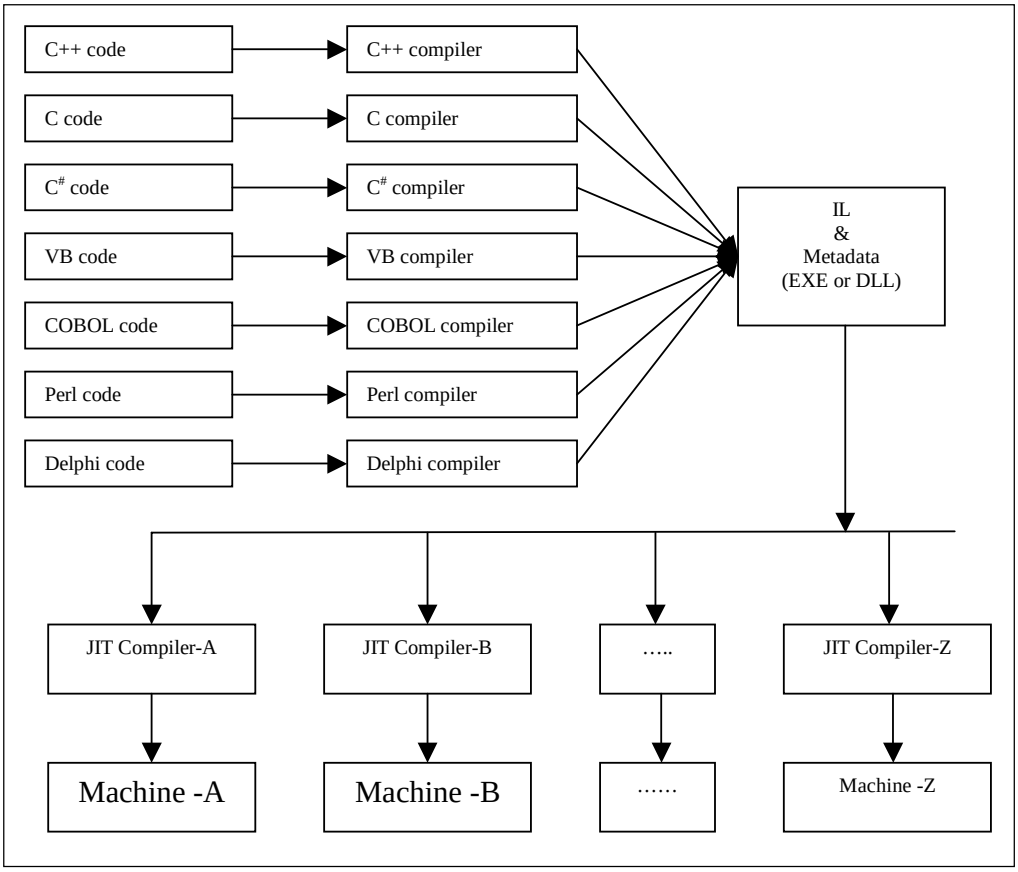
Similarly, compile the following C# code.

```
Namespace math_sum
{
  using System;
  public class ModuleOne
  {
    public ModuleOne(){}
    public int SumIt(int x, int y)
    {
      return(x + y);
    }
    public static int Main(string[] args)
    {
      ModuleOne c = new ModuleOne();
        int x1 = c.SumIt (23, 10);
        Console.WriteLine("23 + 10 = {0}.", x1);
        Return 0;
    }
  }
}
```

The dump of its IL looks like this.

```
.method public hidebysig instance int32 SumIt(int32 x, int32 y) il
managed
{
  //Code size 8 (0x8)
  .maxstack2
  .locals  ([0] int32 V_0)
  IL_0000: ldarg.1
  IL_0001: ldarg.2
  IL_0002: add
  IL_0003: stloc.0
  IL_0004: br.s IL_0006
  IL_0006: ldloc.0
  IL_0007: ret
} //end of method ModuleOne::SumIt
```

This IL code can be compiled into native machine code for individual machine on the fly by JIT (Just In Time also called Jitter) compilers provided by .NET runtime as shown below:



.NET platform is highly secured. The runtime is responsible for the security enforcement. For instance, though it allows executable attached to an e-mail can play an animation on screen sing a song, it protects the personal data, file system, or network resources from being accessed.

Conventionally, an executable program used to be stored as a single file as .EXE and .COM or broken into .DLLs. Under .NET platform, the unit of execution and deployment is the assembly.

The assembly is the primary unit of deployment within the .NET Framework. Within the base class libraries is a class that encapsulates a physical assembly appropriately named Assembly. When this book refers to the class or an instance of the class, it will be denoted as Assembly. This class exists in the System namespace. An assembly can contain references to other assemblies and modules. Execution usually begins with an assembly that has an .exe suffix. The application can use shared code by importing the assembly that contains the shared code with an explicit reference. (You can add the reference via the "Add References" node in Visual Studio .NET or include it via a command-line switch /r). The application can also explicitly load an assembly with Assembly.Load or Assembly.LoadFrom.

Once the code is "loaded," execution of the code can begin. In the case of unmanaged code, the compiler and linker have already turned the source into native instructions, so those instructions can begin to execute immediately. Separate compiles must exist for every different native environment.

During the execution of an assembly, the CLR checks the method that is about to be run to examine whether it has been turned into native code. If the method has not been turned into native code, then the code in the method is Just-In-Time compiled (JIT).

The process of loading a method and compiling it if necessary is repeated until either all of the methods in the application have been compiled or the application terminates. The rest of this chapter explores the environment in which the CLR encloses each class method.

Starting a Method

An assembly is a self-contained unit. Apart from the IL version of the program it also contains information about the code itself. This associated information is called metadata. The CLR requires metadata of the assembly to run it. Following are the ingredients the CLR requires to run a method of an assembly.

MSIL Instructions

The compiled version of the program in MSIL format is available in the assembly. The metadata consists of a pointer to the set of code for each method.

Method signature

Each method is identified by a unique each. The signature specifies how the method shall be called including its return type, parameter count and type.

Exception array

Each assembly includes a list of exceptions it throws. The CLR maintains an array filled with metadata about these exceptions. This array contains information such as type of the exception, an offset address to the first instruction after the exception try block, and the length of the try block and the like. It may also contain offset to the handler code, the length of the handler code, and a token describing the class that is used to encapsulate the exception.

Evaluation stack

Recall the ILDASM dump of the program listed in the previous section. The listing describes the size of the evaluation stack. This size is logical size. The size specifies the maximum number of items that would be needed to push on the stack. The physical size of the items and the stack is determined at runtime by the CLR when the code about to be JITted.

Locals array

A logical array is also maintained by the CLR for the number of local variables to be created in the local storage for the method. For each local variable a flag is also stored which specifies with what value the variable shall be initialized.

The CLR compiles all these information into a native stack frame typical to a machine. This native stack frame is subsequently JITted to obtain the native code for that machine. The call to the method is made in such a way as to allow the CLR to have marginal control of the execution of the method and its state. When the CLR calls or invokes a method, the method and its state are put under the control of the CLR in what is known as the Thread of Control.

Types supported by IL

Though each .NET aware language has its own characteristic data types, after compiling into IL code all of the types must be converted into a set of types supported by the IL. The IL instructions assume certain fixed types of data types. Here is the list of IL supported types.

Integer types:

Type	Description
int8	8-bit 2's complement signed value.
unsigned int8 (byte)	8-bit unsigned binary value.
int16 (short)	16-bit 2's complement signed value.
unsigned int16 (ushort)	16-bit unsigned binary value.
int32 (int)	32-bit 2's complement signed value.
unsigned int32 (uint)	32-bit unsigned binary value.
int64 (long)	64-bit 2's complement signed value.
unsigned (ulong)	64-bit unsigned binary value.
native int	Native size 2's complement signed value.
native unsigned int	Native size unsigned binary value.

Float types:

Type	Description
float32 (float)	32-bit IEC 60559:1989 floating point value.
float64 (double)	64-bit IEC 60559:1989 floating point value.
F-Native size	This variable is internal to the CLR and is not visible by the user.

Reference types:

Type	Description
O	Native size object reference to managed memory.
&	Native size managed pointer.

Homes for Values

The CLR has to keep track of the objects that a program creates (and destroys). For this purpose the concept of a home for an object is employed. Simply stated, an object's home is where the value of the object is stored. When an object is passed by reference, it must have a home because the address of the home is passed as a reference. However, two types of data do not have a home - constants and intermediate values on the evaluation stack from IL instructions or return values from methods.

Method Calling Convention

The CLR method calling convention goes like this. If the method being called is a method on an instance, a reference to the object instance is pushed on the stack first, followed by each of the arguments to the method in left-to-right order. The result is that the this pointer is popped off in opposite manner.

In case of a static method, no associated instance pointer exists and the stack contains only the arguments. For the calling instruction, the arguments are pushed on the stack in a left-to-right order followed by the function pointer that is pushed on the stack last. The CLR and the JIT generate most efficient native calling convention following this convention.

Parameter Passing

Generally, there are two different mechanisms in which the CLR performs parameter-passing to and from a method.

By value:

The CLR uses the stack for this type of parameter-passing. Since only a copy of the value is required to be passed, the method is unable to modify the actual parameter being passed. The value being passed is simply pushed onto the stack for built-in types such as integers, floats, and the like. For objects, an O type reference to the object is placed on the stack. For managed and unmanaged pointers, the address is placed on the stack.

For user-defined types, the value is placed on the evaluation stack either directly using ldarg, ldloc, ldfld, or ldsfld instructions or the address of the value is computed and the value is then loaded onto the stack with the ldobj instruction.

By reference:

In this case, the value is not passed instead the address of the parameter is passed to the method. Obviously, this way a method can modify a passed parameter. Note however that only values that have homes can be passed by reference. The homeless variables cannot be passed this way because it is the address of the home that is passed.

Student Activity 2

1. Explain how .NET attains language independence in addition to platform independence
2. How is the .NET framework related with Visual Studio.Net?
3. Explain the mechanisms in which CLR performs parameter passing to and from a method.

1.5 C# PROGRAMMING BASICS

To learn the basics of a programming language, it is always better to write a simple program first. Let us begin with a very simple working program in C#. The program listed below displays - Welcome to the World of C Sharp - on the screen. We will save this program in a text file - FirstProgram and attach a file name extension - cs.

```
/*
This is our first C Sharp program.
The program displays a message on the screen.
*/
class FirstProgram
{
  public static void Main()
  {
    System.Console.WriteLine("Welcome to the World of C Sharp");
  }
}
```

The program can be saved on the disk as a text file - FirstProgram.cs - using any text editor like Microsoft Windows Notepad. The text file of the program is called its source code. In order to run the program you must compile the source code (.cs) into executable code (.exe). The command that compiles this (.cs) file into (.exe) file is csc. Type the following at the DOS console prompt to invoke the compiler.

```
csc /r:System.dll FirstProgram.cs
```

This will create the executable file - FirstProgram.exe. Type the name of the exe file - FirstProgram - at the prompt to run the program. You should see the following output.

Welcome to the World of C Sharp

This is the general method of creating source and executable codes. However, .NET compliant language tools also come into GUI (Graphical User Interface) form. The IDE (Integrated Development Environment) provides easy to use user-friendly editor, compiler, debugger and loader. We will learn the use of IDE in subsequent sections.

Before we move any further, let us examine the different parts of this elementary program that does nothing useful than displaying a message on the screen. The part being discussed is presented in bold fonts.

Comment Line(s)

```
/*
This is our first C Sharp program.
The program displays a message on the screen.
*/
class FirstProgram
{
  public static void Main()
  {
    System.Console.WriteLine("Welcome to the World of C Sharp");
  }
}
```

Like in C and C++ everything enclosed between '/*' and '*/' is considered as a comment. While compilation the compiler ignores the comment. Comments are a useful way of documenting a program.

The comments enclosed between '/*' and '*/' pairs can be extended to multiple lines. This is called block commenting. C# provides yet another comment syntax that comments a single line. Anything written after '//' on a line is treated as a comment. Thus, the block comment of the FirstProgram can also be written as shown below.

```
//This is our first C Sharp program.
//The program displays a message on the screen.
class FirstProgram
{
  public static void Main()
  {
    System.Console.WriteLine("Welcome to the World of C Sharp");
  }
} //End of the program
```

Note that a line comment has also been introduced in the last line of the program.

```
Class Definition
/*
This is our first C Sharp program.
The program displays a message on the screen.
*/
class FirstProgram
{
  public static void Main()
  {
    System.Console.WriteLine("Welcome to the World of C Sharp");
  }
}
```

Like in any other object-oriented language such as C++ and Java, in C# also the main component of a program is a class that encapsulates data and methods. A C# class has the following form.

```
class classname
{
  .....
  .....
}
```

A class in C# is defined using - class - keyword. The calssname is a user-defined identifier and it simply identifies a class. In our example program there is a single class - FirstProgram. All the methods and codes are inside the class within a pair of braces - { - and - }. The braces - { - and - } signify the start and the end of the class definition respectively.

```
The Main Method
/*
This is our first C Sharp program.
The program displays a message on the screen.
*/
class FirstProgram
{
  public static void Main()
  {
    System.Console.WriteLine("Welcome to the World of C Sharp");
  }
}
```

An application (or program) may have many classes and a class may have a number of methods. A method executes only when it is called. A special method of a C# class is the Main method. Main method is the starting point of execution of an application. Therefore, one class in the application must have a Main method. When the application is executed, its Main method executes first automatically. In case your application does not have a class with the Main method, the compiler will report an error message - No Start Point Defined.

The syntax of the Main method is given below.

```
public static void Main()
{
  ...
  ...
}
```

The Main method also has several parts. Let us understand each one of them in brief.

public: A C# method may or may not be accessible to other methods. Accessibility of a method is specified by an access modifier. public is an access modifier that implies that the method is accessible from all the other methods. Since the Main method should be accessible to everyone in order for the .NET Runtime to be able to call it automatically it is always defined as public.

static: This indicated that the method is a Class Method. Hence it can be called without making a instance of the class first.

void: A method is always expected to return a value to its caller. However, when no value is retuned the value returned must be indicated as void type. Nobody stops you form letting your function return any other data type.

Main(): A method may have any valid name and the programmer is free to use any name for his methods. However, the Main method (with capitalized M) has a special meaning. This is the main entry point of the class. Programmers write their executing codes in this method, which may create other objects and call accessible methods on them.

The System.Console.Writeline method

Writeline method is a static member of the Console class of the System namespace. Namespace is a way to package many related classes under one identity. What the method it does is obvious. It prints on the console whatever is provided to it as argument.

As it is clear from the example, writing a C# program is basically writing a class. Every class is derived from the Object class in C#. The base class of every other class in C# is the class Object. It is defined under System namespace. It is the top most class found at the root of the .NET class hierarchy.

Comparison between C++ and C#

Since C# has close resemblance with the C++ syntax, it is only expectable that the two languages will have remarkable similarities. Indeed the two are similar in many ways (see the list below).

Similarities:

Both C# and C++ are object-oriented as also case-sensitive. In both the languages one cannot use a variable before declaring it explicitly. Besides, objects are instantiated from their classes using new keyword.

Dissimilarities:

However, the similarities end here. The points of differences are that while the entry point in C++ is void main (), in C# everything happens in a class. While in C++ a library needs to be explicitly referenced, in C# there is an assembly - mscorlib.dll - which implicitly loads all the required libraries. While C++ uses double colon (:J for a resolution operator, C# uses a period (.). Here is an example.

In C++ MyClass::MyMethod();

In C# MyClass.MyMethod();

Comparison between Java and C#

Similarities:

C# has a few notable similarities with Java. Both implement the concept of automatic garbage collection.

In both the languages, source code is converted to intermediate code - in C# it is converted to MSIL whereas in java it is converted into bytecode. Both use JIT compiler to convert the intermediate code to native code at runtime. Class is the action-joint for both the languages. Neither of the two languages uses pointers or header-files and uninstantiated variables. Exception handling concept is very similar to both the languages.

Both are type safe languages. In both the cases, the Object class is base class for all other classes. Both the languages have their exclusive execution environment - CLR in C# and JVM in Java.

Dissimilarities:

C#, being a newer language than Java, has introduced a number of improvements over Java. The points of differences between the two are therefore due to these improvements.

Java relies on code interpretation while C# is never interpreted. This makes C# code execute faster than that in Java. Java does not support Win32 and COM platforms whereas C# does. Unlike Java structure type, structure data type in C# is value type and not reference type and therefore have no inheritance. C# allows the programmer to mark certain codes to be unsafe to allow it to have memory access. In C# all exceptions are unchecked.

While all the methods in Java are by default virtual, in C# they must to be explicitly declared virtual. C# provides more numerical precision in its data types than that in Java. There is no need of wrapper classes to use the value data types as reference data types in C# unlike Java.

.NET Framework Class Library

What makes the .NET Framework much suitable for the developers is its class library. Class library is a collection of pre-compiled reusable classes and types. These classes and types integrate with the common language runtime. The entire class library has been built on the object-oriented principles just as you would create classes from your base classes. These classes provide types from which you can derive your classes. This makes the .NET Framework types easy to use. It also makes it easier to migrate a programmer from other platform to .NET.

Some of the applications that can be developed on the .NET Framework are listed below.

Console applications:

These are applications meant to be used without GUI. The user interface remains the console (keyboard and monitor).

Windows GUI applications:

These applications use standard Win32 GUI components.

ASP.NET applications:

These applications are highly interactive systems best suited for Internet and intranet environments.

Student Activity 3

1. Explain the different ways of adding comments in a C# program?
2. Illustrate the comment line operators of C# using suitable example.
3. Explain the Main Method with its parts. What happens when an application does not have a class with the Main method?
4. Compare and contrast between C# and C++
5. Compare and contrast between C# and Java.
6. What are some of the applications that can be developed on the .NET Framework?

1.6 SUMMARY

- C# several features and capabilities, which enable programmers to overcome the limitations of other programming languages such as Visual Basic, C and C++ and Java.
- The .Net Framework is a new computing platform designed to simplify application development in the highly distributed environment of the Internet.
- The four components of .Net framework are .NET Common Language Runtime, Common Type, Common Language Specification and Class Library.
- Net is equipped with a common run time environment that can be seamlessly shared by all the .Net aware languages.
- C# unifies the type system by letting you view every type in the language as an object.
- C# is a modern, type safe programming language, object oriented language that enables programmers to quickly and easily built solutions for the Microsoft .Net platform.
- The .Net framework also provides the collection of classes and tools to aid in development and consumption of web services applications.
- In order to use C# and the .Net framework classes, you first need to install either the .Net framework SDK, or else Visual Studio.NET.
- There are two types of comments- one line comment (//) and multiple line comment (/* …*/)

- A special method of a C# class is the main method which is the starting point of execution of an application. It has several parts such as public, static, void and Main()
- What makes the .NET Framework suitable for developers is its class library, which is a collection of pre-compiled reusable classes and types.

1.7 KEYWORDS

.NET platform: It is a programming framework that provides both integrated development environment and a uniform runtime environment and this platform is aptly suited for application development and deployment in highly distributed environment such as the Internet.

Visual studio .Net: It is a Multilanguage collection of programming tools that allow programmers to write programs in different languages.

Common Language Run Time: It is the heart of .Net framework and is responsible for overall execution of a .Net program.

Common Type System (CTS) : It is used by every language built on the .NET Framework., the CTS specifies no particular syntax or keywords, but instead defines a common set of types that can be used with many different language syntaxes.

Common Language Infrastructure (CLI): It is an open specification developed by Microsoft that describes the executable code and runtime environment that form the core of the Microsoft .NET Framework

Namespaces: Net framework base classes are organized in two folders and units in which C# classes are organized are called Namespaces.

1.8 REVIEW QUESTIONS

1. Explain the relevance of the following terms with respect to C#.
   a. Run-anywhere
   b. Objects owned
   c. Lifestyle choice
   d. Type safe
2. Explain how .NET attains language independence in addition to platform independence.
3. Compare and contrast between C#, C++ and Java.
4. What are .NET aware languages?
5. Explain the advantages of C# over C++ and Java.
6. What is the difference between C# and C#.net?
7. What is the difference between C# and .Net?
8. What are the different functionalities associated with CLR?
9. How a Web Service is different from a Web based application?
10. Why is C# is called as backward compatible language?
11. Can a C# class have more than one main method? If yes, what purpose does it serve?
12. What is the purpose of aliasing a namespace?
13. Illustrate the comment line operators of C# using suitable example.
14. Write a class echo that heads your name from keyboard and prints it back on the monitor.
15. Describe different mathematical function available in Math namespace.

16. List down the advantages and limitations of using Aliases.
17. Write the syntax of declaring an alais.
18. Which mathematical function is used to check whether a number is positive or negative?

1.8 FURTHER READINGS

CLR via C#, Second Edition (Paperback) by Jeffrey Richter, Publisher: Microsoft Press; 2nd edition (February 22, 2006)

Essential C# 2.0 (Microsoft .NET Development Series) (Paperback) by Mark Michaelis, Publisher: Addison-Wesley Professional (July 13, 2006)

C# Programming Language, The (2nd Edition) (Microsoft .NET Development Series) (Hardcover) by Anders Hejlsberg, Scott Wiltamuth, Peter Golde, Publisher: Addison-Wesley Professional; 2 edition (June 9, 2006)

Programming in the Key of C# (Paperback) by Charles Petzold, Publisher: Microsoft Press; 1 edition (July 30, 2003)

UNIT

2

C# DATA TYPES

L E A R N I N G O B J E C T I V E S

After studying this unit, you should be able to:

- Correct number and person of verb in a sentence.
- Avoid mistakes of verbs regarding their being singular/plural.

U N I T S T R U C T U R E

2.1 INTRODUCTION

A datum (singular of data) is a value in the context of processing. Data can assume one among various different forms. The type of processing that can be operated on a datum determines. Thus a data of numeric type allows arithmetic operation on itself while a data of text type does not. Note that the data does not depend on the data itself. It depends on the operations one wants to put the data to. For instance, a phone number may be taken as text type instead of numeric type since one would not subject a phone number for addition, subtraction or any other arithmetical operation.

Data is usually stored on storage devices like a hard disk or a compact disk. However, for processing purposes it must be brought and stored in the main memory (RAM). A variable, which could be referenced by a suitable name, is employed to store data in the RAM. Note that the type of the variable provides the necessary semantics to the value stored in it. Thus, the value 5 stored in a variable of integer type is different from the value 5 stored in a character type variable.

When a variable is created in the memory by a program it needs to know what type of data it is supposed to store. For this reason every programming language defines various types of data it supports and provides means to create and manipulate them.

Data type thus determines following aspects of a variable:

1. The domain of values it can store
2. The operations that can be applied on the data stored in the variable
3. The size (in terms of number of bytes) of the variable in the memory

Programs are primarily written to process data. Therefore, a programming language must provide a variety of data types to support processing of different types of data the users may have. Data types fall into two different categories:

- *Primitive data types:* Primitive data types are the fundamental data types which cannot be divided into simpler data types. For this reason, they are also known as atomic data types. The range of the values assignable to primitive data types as also their size and operations are pre-defined in the language itself.
- *User defined data types:* Primitive data types are not sufficient for many programming problems. Therefore, a programming language also provides mechanism to create new data types by modifying and grouping primitive data types. These data types, which are defined by programmers, are called user defined data types.

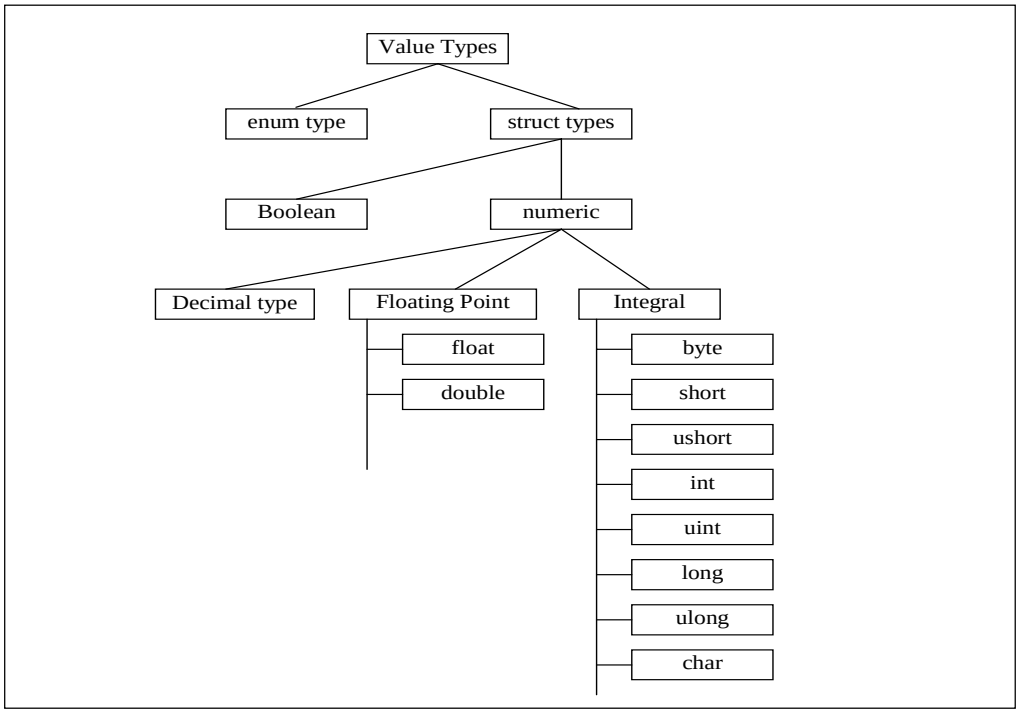
Different data types can be used in three different forms - value type, reference type and pointer type.

- *Value type:* In this form the data type stores real data. When the data are queried by different function a local copy of these memory cells are created. It guarantees that changes made to data in one function don't change them in some other function.
- *Reference type:* Reference type data do not store the actual data. They store reference which indirectly access a data.
- *Pointer type:* Pointer type does not exist in C#. However, it does provide mechanism to use pointer types in unsafe code.

The most important improvement in C# data type system is that it is unified. A value of any type can be treated as an object. Every type in C# directly or indirectly derives from the object class type, the root base class for all other classes. Values of reference types are treated as objects simply by viewing the values as type object. Values of value types can be treated as objects as well as mere values by performing boxing and unboxing operations.

2.2 VALUE TYPES

C# provides two categories of value types - struct type or enumeration type. A set of predefined struct types is also available. It is called the simple types. The simple types are identified through reserved words, and are further subdivided into numeric types, integral types, and floating-point types.



Value types are also classes. They implicitly inherit from class object. However, value types are sealed and cannot be extended to form sub-classes. A variable of a value type always contains a value of that type. Unlike reference types, it is not possible for a value of a value type to be null or to reference an object of a more derived type.

Assignment to a variable of a value type creates a copy of the value being assigned. This differs from assignment to a variable of a reference type, which copies the reference but not the object identified by the reference.

All value types implicitly declare a public parameterless constructor called the default constructor. The default constructor returns a zero-initialized instance known as the default value for the value type. For an enum-type E, the default value is 0. For a struct-type, the default value is the value produced by setting all value type fields to their default value and all reference type fields to null. Like any other constructor, the default constructor of a value type is invoked using the new operator.

Consider the code listed below. Here both i and j variables are initialized to zero.

```
class TestOne
{
  void TestFunction()
{
     int i = 0;
     int j = new int();
  }
}
```

Since every value type implicitly has a public parameterless constructor, it is not possible for a struct type to contain an explicit declaration of a parameterless constructor. A struct type is however permitted to declare parameterized constructors (see the code listed below).

```
struct Point3D
{
  int x, y, z;
  public Point3D(int x, int y, int z)
  {
     this.x = x;
     this.y = y;
     this.z = z;
  }
}
```

Having defined this, look at the following statements.

Point3D p1 = new Point3D();

Point3D p2 = new Point3D(0, 0, 0);

These statements create Point3D objects p1 and p2 with x, y and z initialized to zero.

Structure types

A structure (struct) type is a value type that can declare constructors, constants, fields, methods, properties, indexers, operators, and nested types.

Simple types

C# provides a set of predefined struct types called the simple types. The simple types are identified through reserved words, but these reserved words are simply aliases for predefined struct types in the System namespace, as described in the table below.

Reserved word	Aliased type
Sbyte	System.SByte
Byte	System.Byte
Short	System.Int16
Ushort	System.UInt16
Int	System.Int32
Uint	System.UInt32
Long	System.Int64
Ulong	System.UInt64
Char	System.Char
Float	System.Single
Double	System.Double
Bool	System.Boolean
Decimal	System.Decimal

Because a simple type aliases a struct type, every simple type has members. For example, int has the members declared in System.Int32 and the members inherited from System.Object, and the following statements are permitted:

```
int i = int.MaxValue;    // System.Int32.MaxValue constant
string s = i.ToString(); // System.Int32.ToString() instance method
string t = 123.ToString();  // System.Int32.ToString() instance method
```

The simple types differ from other struct types in that they permit certain additional operations:

- Most simple types permit values to be created by writing literals. For example, 123 is a literal of type int and 'a' is a literal of type char. C# makes no provision for literals of other struct types, and values of other struct types are ultimately always created through constructors of those struct types.
- When the operands of an expression are all simple type constants, it is possible for the compiler to evaluate the expression at compile-time. Such an expression is known as a constant-expression. Expressions involving operators defined by other struct types always imply run time evaluation.
- Through const declarations it is possible to declare constants of the simple types. It is not possible to have constants of other struct types, but a similar effect is provided by static read-only fields.
- Conversions involving simple types can participate in evaluation of conversion operators defined by other struct types, but a user-defined conversion operator can never participate in evaluation of another user-defined operator.

Integral types

C# offers several integral data type, each different in their size and range of values it can store. The various integral types, their sizes and range of values are:

Data type	Size (in bytes)	Range of values
sbyte	1	-128 to 127
byte	1	0 to 255
short	2	-32768 to 32767
ushort	2	0 to 65535
int	4	-2147483648 to 2147483647
uint	4	0 to 4294967295
long	8	9223372036854 to 9223372036854775807
ulong	8	0 to 18446744073709551615
char*	2	0 to 65535

* Though char type is classified as an integral type, it differs from the other integral types in the following ways:

- No implicit conversions exist from other types to the char type. In particular, even though the sbyte, byte, and ushort types have ranges of values that can be fully represented using the char type, implicit conversions from sbyte, byte, or ushort to char are not allowed.
- char type constants must be written as character-literals. Character constants can only be written as integer-literals in combination with a cast. For example, (char)10 is the same as '\x000A'.

Floating point types

Two floating types are supported by C# - float and double. The float and double types are represented using the 32-bit single-precision and 64-bit double-precision IEEE 754 formats, which provide the following sets of values:

Positive zero and negative zero:

In most situations, positive zero and negative zero behave identically as the simple value zero, but certain operations distinguish between the two.

Positive infinity and negative infinity:

Infinities are produced by such operations as dividing a non-zero number by zero. For example 1.0 / 0.0 yields positive infinity, and -1.0 / 0.0 yields negative infinity.

The Not-a-Number value:

Not-a-number values (NaN's) are produced by invalid floating-point operations, such as dividing zero by zero.

The following observations are worth taking a note:

- The float type can represent values ranging from approximately $1.5x10^{-45}$ to $3.4x10^{38}$ with a precision of 7 digits.
- The double type can represent values ranging from approximately $5.0x10^{-324}$ to $1.7x10^{308}$ with a precision of 15-16 digits.
- If one of the operands of a binary operator is of a floating-point type, then the other operand must be of an integral type or a floating-point type, and the operation is evaluated as follows:
  - ❖ If one of the operands of is of an integral type, then that operand is converted to the floating-point type of the other operand.
  - ❖ If a floating-point operation is invalid, the result of the operation becomes NaN.
  - ❖ If one or both operands of a floating-point operation is NaN, the result of the operation becomes NaN.

The bool type

C# does not use numerical values for representing true and false Boolean values like C. There is bool type data that represents boolean logical quantities in C# with only two possible values - true and false. No conversion exists between bool type and other data types.

It is important to note that C# differs from C and C++ in this regard. In C and C++ languages, a zero integral value or a null pointer can be converted to the boolean value false, and a non-zero integral value or a non-null pointer can be converted to the boolean value true. In C#, such conversions are accomplished by explicitly comparing an integral value to zero or explicitly comparing an object reference to null.

Thus the following code is incorrect in C#

```
bool one = 0;

bool two = -4;

bool three = 1;
```

The following assignments are only valid assignments in C#

```
bool one = true;

bool two = false;
```

Also, while in C and C++ is correct,

```
a = TRUE;

if( a ) { };
```

the same is incorrect in C#. In C# one has to explicitly include logical operator as shown below.

```
a= true;

If( a != true) { };
```

Enumeration types

An enumeration type is a distinct type with named constants. Every enumeration type has an underlying integer type. Enumeration types are defined through enumeration declarations as shown below.

```
enum Teams
{
   India, England, Australia, Pakistan
}
```

The keyword enum declares an enum type. The above declaration declares an enum type named Teams whose members are India, England, Australia and Pakistan.

Each enum type has a corresponding integral type called the underlying type of the enum type. This underlying type must be able to represent all the enumerator values defined in the enumeration. An enum declaration may explicitly declare an underlying type of byte, sbyte, short, ushort, int, uint, long or ulong. Note that char cannot be used as an underlying type. An enum declaration that does not explicitly declare an underlying type has an underlying type of int.

Look at the following code.

```
enum Color: long
{
  Red,
  Green,
  Blue
}
```

This code declares an enum with an underlying type of long. A programmer might choose to use an underlying type of long for greater range than that of integer.

The body of an enum type declaration defines zero or more enum members, which are the named constants of the enum type. No two enum-members can have the identical name.

Each enum member has an associated constant value. The type of this value is the underlying type for the containing enum. The constant value for each enum member must be in the range of the underlying type for the enum. Consider the following code.

```
enum Direction: uint
{
  East = -1,
  West = -2,
  North = -3,
  South = -4;
}
```

The above code has a declaration error. The constant values -1, -2, -3 and -4 are not in the range of the underlying integral type uint.

More than one members of an enum type may share the same associated value as shown in the code listed below.

```
enum Direction
{
  East,
  West,
  North,
  Sunrise = East
}
```

Here, two enum members - East and Sunrise have the same associated value, i.e., East. The individual members of the enum type are accessed as usual by dot operator as shown below.

```
  Direction.East
  Direction.West etc.
```

You can assign the associated value to an enum either implicitly or explicitly. If the declaration of the enum member has a constant-expression initializer, the value of that constant expression, implicitly converted to the underlying type of the enum, is the associated value of the enum member. If the declaration of the enum member has no initializer, its associated value is set implicitly. If the member is the first enum member declared in the enum type, its associated value is zero. Otherwise, the associated value of the enum member is obtained by increasing the associated value of the previous enum member by one. This increased value must be within the range of values that can be represented by the underlying type.

Consider the following code.

```
using System;
enum Direction
{
  East,
  West = 15,
  North,
  South
}
class Test
{

  static void Main()
{
  Console.WriteLine(ShowDirection(Direction.East));
  Console.WriteLine(ShowDirection(Direction.West));
  Console.WriteLine(ShowDirection(Direction.North));
  Console.WriteLine(ShowDirection(Direction.South));
}
  static string ShowDirection(Direction c)
  {
    switch (c)
    {
      case Direction.East:
        return String.Format("East = {0}", (int)c);
      case Direction.West:
        return String.Format("West = {0}", (int)c);
      case Direction.North:
        return String.Format("North = {0}", (int)c);
      case Direction.South:
        return String.Format("South = {0}", (int)c);
      default:
        return "Invalid direction";
    }
  }
}
```

This program prints out the enum member names and their associated values. The output is:

```
East = 0
West = 15
North = 16
South = 17
```

Note that the enum member East is automatically assigned the value zero since it has no initializer and is the first enum member. The enum member West is explicitly given the value 15 and the enum members North and South are automatically assigned the value one greater than the member that textually precedes it, i.e., 16 and 17.

The associated value of an enum member may not, directly or indirectly, use the value of its own associated enum member. Other than this circularity restriction, enum member initializers may freely refer to other enum member initializers, regardless of their textual position. Within an enum member initializer, values of other enum members are always treated as having the type of their underlying type, so that casts are not necessary when referring to other enum members.

Consider the following code.

```
enum CircularExample
{
   One = Two,
   Two
}
```

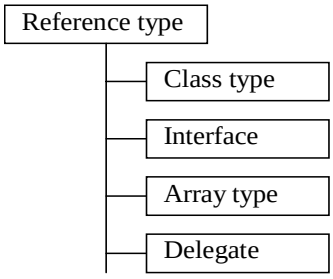
This code is invalid because the declarations of One and Two are circular. One depends on Two explicitly, and Two depends on One implicitly. Enum members are named and scoped in a manner exactly analogous to fields within classes. The scope of an enum member is the body of its containing enum type. Within that scope, enum members can be referred to by their simple name. From all other code, the name of an enum member must be qualified with the name of its enum type. Enum members do not have any declared accessibility-an enum member is accessible if its containing enum type is accessible.

Student Activity 1

1. Define data. What are the categories of data types?
2. Explain the different forms that data types can take.
3. Explain the two categories of value types C# provides.
4. What is the difference between values types and reference types?
5. Explain the concept of Enumeration with the help of suitable example.
6. What are the advantages of enumeration data type?

2.3 REFERENCE TYPES

A reference type is a class type, an interface type, an array type, or a delegate type.



A reference type value is a reference to an instance of the type, the latter known as an object. The special value null is compatible with all reference types and indicates the absence of an instance.

Class types

A class type defines a data structure that contains data members (constants, fields, and events), function members (methods, properties, indexers, operators, constructors, and

destructors), and nested types. Class types support inheritance, a mechanism whereby derived classes can extend and specialize base classes. Instances of class types are created using object-creation-expressions.

Object class type

It is the ultimate base class of all other types. Every type in C# directly or indirectly derives from the object class type.

The object keyword is simply an alias for the predefined System.Object class. Writing the keyword object is exactly the same as writing System.Object, and vice versa.

String type

The string type is a sealed class type that inherits directly from object. Instances of the string class represent Unicode character strings. Values of the string type can be written as string literals.

The string keyword is simply an alias for the predefined System.String class. Writing the keyword string is exactly the same as writing System.String, and vice versa.

A character that follows a backslash character (\) in a regular-string-literal-character must be one of the following characters: ', ", \, 0, a, b, f, n, r, t, u, U, x, v. Otherwise, a compile-time error occurs.

Following are some examples of valid string literals.

```
string a = "hello, there";

string b = @"hello, there";

string c = "hello \t there";

string d = @"hello \t there";

string e = "Mahesh said \"Hello\" to me";

string f = @"Mahesh said ""Hello"" to me";

string g = "\\\\nic\\share\\file.txt";

string h = @"\\nic\share\file.txt";

string i = "first\nsecond\nthird";

string j = @"first second third";
```

The last string literal, j, is a verbatim string literal that spans multiple lines. The characters between the quotation marks, including white space such as newline characters, are preserved verbatim.

Note that each string literal does not necessarily result in a new string instance. When two or more string literals that are equivalent according to the string equality operator appear in the same program, these string literals refer to the same string instance (see the following program).

```
class TestTwo

{

  static void Main()

  {

    object a = "jumbo";

    object b = "jumbo";

    Console.WriteLine(a == b);

  }

}
```

The output is True because the two literals refer to the same string instance.

The string class derives directly from the object, and is sealed i.e. user cannot derive from it. The string keyword is simply an alias for the predefined System.String class.

String Manipulation in C#

System.String provides a number of properties and methods through which strings may be manipulated in various ways. One of the properties is

Length: This property returns the length of the current string.

Some of the useful string-methods are:

Concat(): This is a static method of String class which returns a new string by concatenating (joining) two strings.

CompareTo(): Compares two strings.

Copy(): Returns a new copy of the existing string.

Format(): This method is used to format the string.

Insert(): This method inserts a string into another string.

PadLeft(): This method appends a string with some character to the left side.

PadRight(): This method appends a string with some character to the rightside.

Remove(): Removes a substring from a string.

Replace(): Replaces a substring with some other substring.

ToUpper(): Coverts to upper case.

ToLower(): Converts to lower case.

Although, string data type is reference type, the equality operator (== and !=) compare the string objects and not the memory they reference. Also, addition operator (+) has been overloaded as concatenating operator for strings.

String arrays are indexed as will be evident from the following example.

```
using System;
class TestThird
{
  public static void Main()
  {
    string[] names = {"Rajesh", "Vibhor", "Geeta", "Jyoti"};
    foreach (string person in names)
    {
      Console.WriteLine("{0} ", person);
    }
  }
}
```

Within the "foreach" parentheses is an expression composed of two elements divided by the keyword "in". The right-hand side is the collection you want to use to access each element. The left-hand side holds a variable with a type identifier compatible with whatever type the collection returns.

Every time through this loop the collection is queried for a new value. As long as the collection can return a value, this value will be put into the read-only variable and the expression will return true, thus causing the statements in the "foreach" block to be executed. When the collection has been fully traversed, the expression will evaluate to false and control will transfer to the first executable statement following the end of the "foreach" block.

Interface types

An interface defines a contract. A class or struct that implements an interface must adhere to its contract. An interface may inherit from multiple base interfaces, and a class or struct may implement multiple interfaces.

Array types

An array is a data structure that contains a number of variables which are accessed through computed indices. The variables contained in an array, also called the elements of the array, are all of the same type, and this type is called the element type of the array.

Delegate types

A delegate is a data structure that refers to a static method or to an object instance and an instance method of that object.

The closest equivalent of a delegate in C or C++ is a function pointer, but whereas a function pointer can only reference static functions, a delegate can reference both static and instance methods. In the latter case, the delegate stores not only a reference to the method's entry point, but also a reference to the object instance for which to invoke the method.

2.4 OPERATORS

Expressions are constructed from operands and operators. The operators of an expression indicate which operations to apply to the operands. Examples of operators include +, -, *, / , and new. Examples of operands include literals, fields, local variables, and expressions.

There are three types of operators:

- Unary operators The unary operators take one operand and use either prefix notation (such as -x) or postfix notation (such as x++).
- Binary operators The binary operators take two operands and all use infix notation (such as x + y).
- Ternary operator Only one ternary operator, ?:, exists. The ternary operator takes three operands and uses infix notation (c? x: y).

The order of evaluation of operators in an expression is determined by the precedence and associativity of the operators. Certain operators can be overloaded. Operator overloading permits user-defined operator implementations to be specified for operations where one or both of the operands are of a user-defined class or struct type.

Operator Precedence And Associativity

When an expression contains multiple operators, the precedence of the operators control the order in which the individual operators are evaluated. For example, the expression x + y * z is evaluated as x + (y * z) because the * operator has higher precedence than the + operator. The precedence of an operator is established by the definition of its associated grammar production. For example, an additive-expression consists of a sequence of multiplicative-expressions separated by + or - operators, thus giving the + and - operators lower precedence than the *, /, and % operators.

The following table summarizes all operators in order of precedence from highest to lowest:

CATEGORY	OPERATORS
Primary	(x) x.y f(x) a[x] x++ x-- newtypeof sizeof checked unchecked
Unary	+ - ! ~ ++x --x (T)x
Multiplicative	* / %
Additive	+ -
Shift	<< >>
Relational	< > <= >= is as
Equality	== !=
Logical AND	&
Logical XOR	^
Logical OR	|
Conditional AND	&&
Conditional OR	||
Conditional	?:
Assignment	= *= /= %= += -= <<= >>= &= ^= |=

When an operand occurs between two operators with the same precedence, the associativity of the operators controls the order in which the operations are performed:

- Except for the assignment operators, all binary operators are left-associative, meaning that operations are performed from left to right. For example, x + y + z is evaluated as (x + y) + z.
- The assignment operators and the conditional operator (?:) are right-associative, meaning that operations are performed from right to left. For example, x = y = z is evaluated as x = (y = z).
- Precedence and associativity can be controlled using parentheses. For example, x + y * z first multiplies y by z and then adds the result to x, but (x + y) * z first adds x and y and then multiplies the result by z.

Student Activity 2

1. Explain reference types.
2. List down some of the main methods available in String class.
3. What are the advantages of using string type rather than character array type variable?
4. What is meant by operators' precedence and associativity?
5. What are the different classes of operators available in C#?

2.5 VARIABLES

During its operation a computer program has to store data values in the memory. The memory location where a data value can be stored and retrieved is called a variable. The interpretation of the value stored in a variable is determined by its type. The C# compilers are designed to check the data compatibility at the compile time itself. C# compiler guarantees that values stored in variables are always of the appropriate type. For this reason C# is called a type-safe language.

The value stored in a variable keeps on changing as the processing progresses. The value can be changed through assignment or through increment (++) and decrement (??) operators. Variable must have some value stored in it before its value can be obtained.

Variable categories

For convenience C# variables can be put into the following categories.

Static variables

A static variable is created when the type in which it is declared is loaded, and ceases to exist when the program terminates. The modifier static before the declaration indicates that the variable is static.

The initial value of a static variable is the default value of the variable's type. For purposes of definite assignment checking, a static variable is considered initially assigned.

Instance variables

A field declared without the static modifier is called an instance variable. An instance variable of a class comes into existence when a new instance of that class is created, and ceases to exist when there are no references to that instance and the instance's destructor (if any) has executed. The initial value of an instance variable of a class is the default value of the variable's type. For purposes of definite assignment checking, an instance variable of a class is considered initially assigned.

An instance variable of a structure has exactly the same lifetime as the structure variable to which it belongs. The initial assignment state of an instance variable of a struct in the same as that of the containing struct variable. In other words, when a struct variable is considered initially assigned, so too are its instance variables, and when a struct variable is considered initially unassigned, its instance variables are likewise unassigned.

Array elements

The elements of an array are created when an array instance is created, and cease to exist when there are no references to that array instance. The initial value of each of the elements of an array is the default value of the type of the array elements. For purposes of definite assignment checking, an array element is considered initially assigned.

Value parameters

A parameter declared without a ref or out modifier is a value parameter. A value parameter comes into existence upon invocation of the function member (method, constructor, accessor, or operator) to which the parameter belongs, and is initialized with the value of the argument given in the invocation. A value parameter ceases to exist upon return of the function member. For purposes of definite assignment checking, a value parameter is considered initially assigned.

Reference parameters:

A parameter declared with a ref modifier is a reference parameter. A reference parameter does not create a new storage location. Instead, a reference parameter represents the same storage location as the variable given as the argument in the function member invocation. Thus, the value of a reference parameter is always the same as the underlying variable.

The following definite assignment rules apply to reference parameters.

- A variable must be definitely assigned before it can be passed as a reference parameter in a function member invocation.
- Within a function member, a reference parameter is considered initially assigned.
- Within an instance method or instance accessor of a structure type, the this keyword behaves exactly as a reference parameter of the structure type.

Output parameters

A parameter declared with an out modifier is an output parameter. An output parameter does not create a new storage location. Instead, an output parameter represents the same storage location as the variable given as the argument in the function member invocation. Thus, the value of an output parameter is always the same as the underlying variable. The following definite assignment rules apply to output parameters.

A variable need not be definitely assigned before it can be passed as an output parameter in a function member invocation.

Following a function member invocation, each variable that was passed as an output parameter is considered assigned in that execution path. Within a function member, an output parameter is considered initially unassigned.

Every output parameter of a function member must be definitely assigned before the function member returns. Within a constructor of a structure type, the this keyword behaves exactly as an output parameter of the structure type.

Local variables

A local variable is declared by a local-variable-declaration, which may occur in a block, a for-statement, a switch-statement, or a using-statement. A local variable comes into existence when control enters the block, for-statement, switch-statement, or using-statement that immediately contains the local variable declaration. A local variable ceases to exist when control leaves its immediately containing block, for-statement, switch-statement, or using-statement.

A local variable is not automatically initialized and thus has no default value. For purposes of definite assignment checking, a local variable is considered initially unassigned. A local-variable-declaration may include a variable-initializer, in which case the variable is considered definitely assigned in its entire scope, except within the expression provided in the variable-initializer.

Within the scope of a local variable, it is an error to refer to the local variable in a textual position that precedes its variable-declarator.

Consider the following code example.

```
class Test
{
  public static int i;
  int j;
  void F(int[] Books, int aVar, ref int bVar, out int cVar)
  {
    int k = 1;
    cVar = aVar - bVar++;
  }
}
```

Here in this example:

i is a static variable

j is an instance variable

Books[0] is an array element

aVar is a value parameter

bVar is a reference parameter

cVAr is an output parameter

Student Activity 3

1. What are the different categories of variables? Explain with the help of suitable examples.
2. What is the difference between Instance variable and class variable?
3. How reference parameters are different from value parameters?
4. How output parameter is different from reference parameter?

2.6 TYPE CONVERSIONS

When assigning a value to a variable it may so happen that the type o variable is different from the type of value being assigned. In such cases the assignment may be denied or may be allowed with some data conversion. All the data-types are not totally unrelated. They can be converted form one form to another provided they make the conversion meaningful. There are certain rules that govern these data conversions.

The type-conversions may be grouped into two categories - implicit and explicit. Implicit conversion takes place without the intervention of the programmer at runtime. On the other hand explicit conversion requires the programmer to explicitly indicate that a conversion is intended.

Implicit conversions can occur in a variety of situations, including function member invocations, cast expressions, and assignments. The pre-defined implicit conversions always succeed and never cause exceptions to be thrown. Properly designed user-defined implicit conversions should exhibit these characteristics as well.

Following are the implicit conversions.

Identity conversion:

Here conversions take place from a type to the same type.

Numeric conversions:

In converting one numeric type to another numeric type the normal rule followed is that the numeric types taking less number of bytes in the memory can be converted into ones taking more number o bytes. This causes no loss of value. Such conversions are listed below.

Data type	Can be converted to
sbyte	short, int, long, float, double, decimal
byte decimal	short, ushort, int, uint, long, ulong, float, double,
byte	int, long, float, double, decimal
ushort	int, uint, long, ulong, float, double, decimal
int	long, float, double, decimal
uint	long, float, double, decimal
long	float, double, decimal
ulong	float, double, decimal
char	ushort, int, uint, long, ulong, float, double, decimal
float	double

Note that conversions from int, uint, or long to float and from long to double may cause a loss of precision, but will never cause a loss of magnitude. Other than that implicit numeric conversions never lose any information. There are no implicit conversions to the char type. This in particular means that values of the other integral types do not automatically convert to the char type.

Enumeration conversions:

An implicit enumeration conversion permits the decimal-integer-literal 0 to be converted to any enum-type.

Reference conversions:

References can be converted from one type to another just as any other type does. The general rule that governs this conversion is that a reference of lower type can always reference to higher type. Here is a synopsis of reference type conversion.

Data type	Can be converted to
Reference type	System.Object type
Class type	Class type of the same class, class type of class derived from this class type
Class type	Interface type provided the given class implements the interface
Interface type	Interface of same type, interface to derived type
Array type	Array type of same dimension, System.Array type, System.ICloneable
Delegate type	System.delegate type
Null type	A reference type

In reference conversions, implicit or explicit, the referential identity of the object being converted does not change. A reference conversion may change the type of a value, however, it never changes the value itself.

Expression conversion:

An implicit constant expression conversion permits the following conversions:

Data type	Can be converted to
int type Constant-expression	sbyte, byte, short, ushort, uint, ulong, the range of the value being within these types
ulong constant-expression	ulong, the range being within thus type

The following conversions are classified as explicit conversions:

- All implicit conversions.
- Explicit numeric conversions.
- Explicit enumeration conversions.
- Explicit reference conversions.
- Explicit interface conversions.
- Unboxing conversions.
- User-defined explicit conversions.
- Explicit conversions can occur in cast expressions.

The explicit conversions are conversions that cannot be proved to always succeed, conversions that are known to possibly lose information, and conversions across domains of types sufficiently different to merit explicit notation.

The set explicit conversions includes all implicit conversions. This in particular means that redundant cast expressions are allowed.

Explicit numeric conversions

The explicit numeric conversions are the conversions from a numeric-type to another numeric-type for which an implicit numeric conversion does not already exist:

```
From sbyte to byte, ushort, uint, ulong, or char.

From byte to sbyte and char.

From short to sbyte, byte, ushort, uint, ulong, or char.

From ushort to sbyte, byte, short, or char.

From int to sbyte, byte, short, ushort, uint, ulong, or char.

From uint to sbyte, byte, short, ushort, int, or char.

From long to sbyte, byte, short, ushort, int, uint, ulong, or char.

From ulong to sbyte, byte, short, ushort, int, uint, long, or char.

From char to sbyte, byte, or short.
```

From float to sbyte, byte, short, ushort, int, uint, long, ulong, char, or decimal.

From double to sbyte, byte, short, ushort, int, uint, long, ulong, char, float, or decimal.

From decimal to sbyte, byte, short, ushort, int, uint, long, ulong, char, float, or double.

Because the explicit conversions include all implicit and explicit numeric conversions, it is always possible to convert from any numeric-type to any other numeric-type using a cast expression.

The explicit numeric conversions possibly lose information or possibly cause exceptions to be thrown. An explicit numeric conversion is processed as follows:

For a conversion from an integral type to another integral type, the processing depends on the overflow checking context in which the conversion takes place:

- In a checked context, the conversion succeeds if the source argument is within the range of the destination type, but throws an OverflowException if the source argument is outside the range of the destination type.
- In an unchecked context, the conversion always succeeds, and simply consists of discarding the most significant bits of the source value.

For a conversion from float, double, or decimal to an integral type, the source value is rounded towards zero to the nearest integral value, and this integral value becomes the result of the conversion. If the resulting integral value is outside the range of the destination type, an OverflowException is thrown.

For a conversion from double to float, the double value is rounded to the nearest float value. If the double value is too small to represent as a float, the result becomes positive zero or negative zero. If the double value is too large to represent as a float, the result becomes positive infinity or negative infinity. If the double value is NaN, the result is also NaN.

For a conversion from float or double to decimal, the source value is converted to decimal representation and rounded to the nearest number after the 28th decimal place if required. If the source value is too small to represent as a decimal, the result becomes zero. If the source value is NaN, infinity, or too large to represent as a decimal, an InvalidCastException is thrown.

For a conversion from decimal to float or double, the decimal value is rounded to the nearest double or float value. While this conversion may lose precision, it never causes an exception to be thrown.

Explicit enumeration conversions

The explicit enumeration conversions are:

From sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, or decimal to any enum-type.

From any enum-type to sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, or decimal.

From any enum-type to any other enum-type.

An explicit enumeration conversion between two types is processed by treating any participating enum-type as the underlying type of that enum-type, and then performing an implicit or explicit numeric conversion between the resulting types. For example, given an enum-type E with and underlying type of int, a conversion from E to byte is processed as an explicit numeric conversion from int to byte, and a conversion from byte to E is processed as an implicit numeric conversion from byte to int.

Explicit reference conversions

The explicit reference conversions are:

`From object to any reference-type.`

`From any class-type S to any class-type T, provided S is a base class of T.`

`From any class-type S to any interface-type T, provided S is not sealed and provided S does not implement T.`

`From any interface-type S to any class-type T, provided T is not sealed or provided T implements S.`

`From any interface-type S to any interface-type T, provided S is not derived from T.`

From an array-type S with an element type SE to an array-type T with an element type TE, provided all of the following are true:

- S and T differ only in element type. In other words, S and T have the same number of dimensions.
- Both SE and TE are reference-types.
- An explicit reference conversion exists from SE to TE.
- From System.Array to any array-type.
- From System.Delegate to any delegate-type.
- From System.ICloneable to any array-type or delegate-type.

The explicit reference conversions are those conversions between reference-types that require run-time checks to ensure they are correct.

For an explicit reference conversion to succeed at run-time, the value of the source argument must be null or the actual type of the object referenced by the source argument must be a type that can be converted to the destination type by an implicit reference conversion. If an explicit reference conversion fails, an InvalidCastException is thrown.

Reference conversions, implicit or explicit, never change the referential identity of the object being converted. In other words, while a reference conversion may change the type of a value, it never changes the value itself.

User-defined explicit conversions

A user-defined explicit conversion consists of an optional standard explicit conversion, followed by execution of a user-defined implicit or explicit conversion operator, followed by another optional standard explicit conversion.

Standard conversions

The standard conversions are those pre-defined conversions that can occur as part of a user-defined conversion.

Standard implicit conversions

The following implicit conversions are classified as standard implicit conversions:

- Identity conversions
- Implicit numeric conversions
- Implicit reference conversions
- Boxing conversions
- Implicit constant expression conversions

The standard implicit conversions specifically exclude user-defined implicit conversions.

Standard explicit conversions

The standard explicit conversions are all standard implicit conversions plus the subset of the explicit conversions for which an opposite standard implicit conversion exists. In other words, if a standard implicit conversion exists from a type A to a type B, then a standard explicit conversion exists from type A to type B and from type B to type A.

User-defined conversions

C# allows the pre-defined implicit and explicit conversions to be augmented by user-defined conversions. User-defined conversions are introduced by declaring conversion operators in class and struct types.

Permitted user-defined conversions

C# permits only certain user-defined conversions to be declared. In particular, it is not possible to redefine an already existing implicit or explicit conversion. A class or struct is permitted to declare a conversion from a source type S to a target type T only if all of the following are true:

- S and T are different types.
- Either S or T is the class or struct type in which the operator declaration takes place.
- Neither S nor T is object or an interface-type.
- T is not a base class of S, and S is not a base class of T.

Student Activity 3

1. What are the different ways to carry out type conversion in C#?
2. Give the sequence of the type casting rules that are applied while evaluating an expression.
3. What is the difference between implicit conversion and explicit conversion?

2.7 SUMMARY

- C# data type is unified. A value of any type can be treated as an object.
- Values of reference type can be treated as objects by simply viewing the values as type object.
- Values of values types can be treated as objects as well as mere values by performing boxing and unboxing operations
- C# provides two categories of value types-struct type or enumeration type.
- If no explicit initialization of variables is performed by the programmers, C# places default values to the variables.
- An operator is a symbol that tells C# to perform some of the operations, actions on one or more operands.
- Value can be assigns to more than one variable at the same time. Example a=b=0
- When an expression contains multiple operators, the precedence of the operator controls the order in which the individual operator is evaluated.
- Data types can be converted from one form to another by typecasting.

2.8 KEYWORDS

Datum: A datum, singular of data, is a value in the context of processing.

Value type: .NET type containing actual data rather than a reference to data stored elsewhere in memory. Simple value types include Boolean, character, decimal, floating point, and integer

Reference type: Variable, which holds a reference (pointer) to data rather than containing the actual data. Reference types include array, class, delegate, and interface.

Operators: C# provides a large set of operators, which are symbols that specify which operations to perform in an expression. C# predefines the usual arithmetic and logical operators, as well as a variety of others.

Variable: It is a storage location in the memory and like a container that holds the value.

Instance variable: Variables that are declared inside the class declaration and outside the scope of any method are called instance variables.

Static Variables: Variables that are declared with the word static are called static variables where in all the objects of the class shares the only one copy of the variables.

Boxing: To convert value type data type in to reference type data type is called Boxing.

Unboxing: To convert reference data type to value type data type is called Unboxing.

Implicit type conversion: The type conversions that are performed automatically by C# compiler are called implicit conversions.

Explicit Type Conversions: The type conversions that are performed with the programmer's intervention are called as explicit conversions.

2.8 REVIEW QUESTIONS

1. Can there be a working C# program without a single Main method? Justify your answer.
2. Differentiate between instance and static variables using a suitable example.
3. What are the different types of data types available in C#?
4. Explain the concept of reference parameter and output parameters with suitable examples.
5. What is the .net framework system type of the data types available iin c#?

2.10 FURTHER READINGS

CLR via C#, Second Edition (Paperback) by Jeffrey Richter, Publisher: Microsoft Press; 2nd edition (February 22, 2006)

Essential C# 2.0 (Microsoft .NET Development Series) (Paperback) by Mark Michaelis, Publisher: Addison-Wesley Professional (July 13, 2006)

C# Programming Language, The (2nd Edition) (Microsoft .NET Development Series) (Hardcover) by Anders Hejlsberg, Scott Wiltamuth, Peter Golde, Publisher: Addison-Wesley Professional; 2 edition (June 9, 2006)

Programming in the Key of C# (Paperback) by Charles Petzold, Publisher: Microsoft Press; 1 edition (July 30, 2003)

UNIT

3

ARRAYS AND STRUCTURES

L E A R N I N G O B J E C T I V E S

After studying this unit, you should be able to:

- Correct number and person of verb in a sentence.
- Avoid mistakes of verbs regarding their being singular/plural.

U N I T S T R U C T U R E

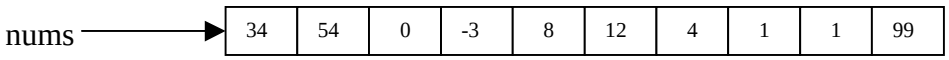
3.1 INTRODUCTION

This chapter will deal with arrays and structures of C# programming. An array is a data structure that contains a number of variables, called the elements of the array, having same name, same type and same size but containing possibly different values. In this chapter you will learn about C# arrays, single-dimensional array, multidimensional array, how they are accessed through indexes and how to go about creating arrays. In C#, array types are reference types derived from the abstract base type System.Array. The latter half of the chapter deals with structures, which are a group of basic data type variables, which behaves as a unit. Further the structure will be contrasted with classes which structures resembles.

3.2 ARRAYS

If you have some experience in computer programming, then irrespective of the language(s) you used you would be well aware of arrays. An array is a data structure that contains a number of identical variables (variables having same name, same type and same size) accessible by a number called index. The number of elements an array has is called its rank.

Following is an array of 10 integer variables or simply put, an array of 10 integers.



Lets us understand different parts of this array.

1. Name of the array: nums. This is the name of the array. It could be any valid identifier allowed by a particular language.
2. The rank of the array: 10, i.e., it has 10 elements.
3. Type of the array: integer, i.e., its elements can store only one integer value each.

4. Indices of the array: 1, 2, 3, 4, 5, 6, 7, 8, 9 and 10. Thus the 10 element of the array are:

   nums[1] nums[2] nums[3] nums[4] nums[5]
   nums[6] nums[7] nums[8] nums[9] nums[10]

   However, in some languages like the language C, the indexing begins at 0 instead of 1. In that case the elements will be referred to as:

   nums[0] nums[1] nums[2] nums[3] nums[4]
   nums[5] nums[6] nums[7] nums[8] nums[9]

   Still in some cases both the options are available as in Visual Basic wherein it could begin with either 0 or 1. The programmer has to specify the option in the beginning of the program. Even square brackets around the indices are language specific. In some languages (such as Visual Basic) parentheses are used in place of square brackets, as shown below.

   nums(0) nums(1) nums(2) nums(3) nums(4)
   nums(5) nums(6) nums(7) nums(8) nums(9)

   Also note that it is not necessary that the indices are always integers. They could be any other variable or expression that evaluate to a valid index value. An index value is valid for a particular array if its value lies between the lower and upper index values exclusive. Thus, if ind is an integer variable whose current value is 3 then all the following are valid index value for the array given above provided the indices fall between the upper and lower index values inclusively.

   ```
   nums[ind]
   nums[ind + 4]
   nums[ind - 2]
   nums[2 * ind]
   nums[ind++]
   ```

5. Values stored in the array in the example are: 34, 54, 0, -3, 8, 12, 4, 1, 1, 99. Assuming that the first index is 0, nums[0] stores 34, nums[5] stores 12 while nums[9] stores 99.

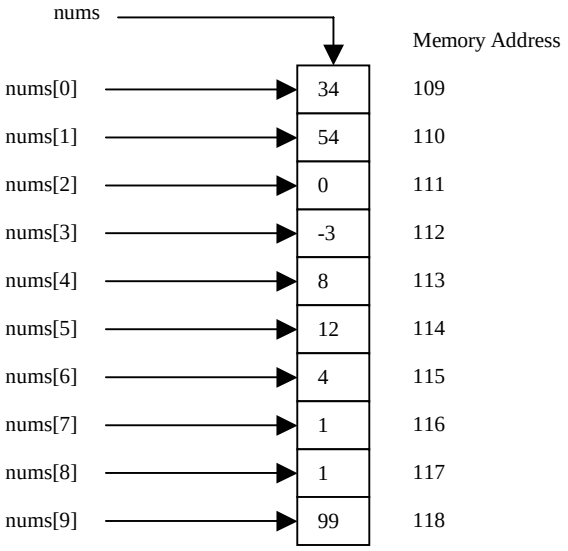
6. These elements behave just as any other integer type variable does. Thus, all the following expressions are valid.

   nums[0] + 3 ;which evaluates to 37
   nums[4] - nums[5] ;which evaluates to -4

   In fact, even the following expression is valid!

   nums[nums[2]] + 4 ;which evaluates to 38

7. The array elements are created at compile time implying that you cannot change the number of elements in the array dynamically at runtime. This is referred to as static allocation of memory. Moreover the elements are allocated contiguous memory locations when created. Thus, assuming that the first element of the above array is allocated memory location - (say) 109, then the array will have the following memory map.



One of the important implications of contiguous memory allocation of the array elements is that the array elements can be manipulated with the help of pointers.

Pointer is a variable (or constant) that is capable of storing memory address of another variable. The name of the array itself is a pointer, a constant pointer to be exact, which stores the address of the first element. In the foregoing example, nums is a pointer type constant that stores 109, the address of the first element of the array. Thus, all the following references are valid.

```
nums                ;points to the first array element
nums+1              ; points to the second array element
```

It is interesting to note that in most of the languages strings of characters are stored in this manner.

8. The number of integers used in the index to access an element of the array is called its dimension. Since in our case (the array nums) only one integer appears into the index, it is one-dimensional array. Arrays may be two-dimensional, three-dimensional or n-dimensional in which case it is called multi-dimensional array.

Creating an array

Array instances are created using array-initializer. While creating an array instance the rank and length of each dimension are specified explicitly which cannot be changed as long as the array exists. Elements of arrays created thus are always initialized to their default value.

Various valid forms of array creation syntax are given below.

Example: Create a one-dimensional array named - intArr - of 5 integers each element containing default values.

```
int[ ] intArr = new int[5];
```

Example: Create a one-dimensional array named - names - of 15 strings each element containing default values.

```
string[ ] names = new string[15];
```

The array-initializer in this case is - int[ ] and string[ ]. The reserved word - *new* - creates an array each and assigns the references to *intArr* and names respectively. Here, in this example, the data type (int), number of elements (5) and dimension (1) are all specified along with the name of the array.

The elements of this array may be initialized with desired integers now explicitly as shown below.

```
intArr[0] = 4    intArr[1] = 6    intArr[2] = 1    intArr[3] = 9    intArr[4] = 3
```

However, one can initialize the array with given values even at the time of creation in the declaration itself. See the following example.

Example: Create a one-dimensional array named - aaa - of integers containing 5 given

Integers - 4, 6, 1, 9, 3.

```
int[ ] aaa = new int[ ] { 4,  6,  1,  9, 3 };
```

Or simply,

```
int[ ] aaa = { 4,  6,  1,  9, 3 };
```

When an array is created with values assigned to its elements, the number of elements need not be explicitly specified as is shown in the example given above. The expressions initialize array elements in increasing order, starting with the element at index zero. The number of

elements in the array-initializer determines the length of the array instance being created. In the example given above the length is 5 (indices from 0 to 4).

Creating a multi-dimensional array is similar to creating one-dimensional array. The array initializer specifies as many levels of nesting as there are dimensions in the array. The number of elements in each of the dimensions may be explicitly stated as shown in the example below.

Example: Create a two-dimensional array (say, 4 rows and 3 columns) named intArr of integers with default values.

int[,] intArr = new int[4, 3];

The indices of this array vary from 0 to 3 for rows and from 0 to 2 for columns. Different values can now be stored in the array by assigning the given values to the individual elements, as shown below.

intArr[0, 0] = 11	intArr[0, 1] = 3
intArr[1, 0] = 5	intArr[1, 1] = 2
intArr[2, 0] = 4	intArr[2, 1] = 9
intArr[3, 0] = 14	intArr[3, 1] = 7

In multi-dimensional arrays the outermost nesting level corresponds to the leftmost dimension and the innermost nesting level corresponds to the rightmost dimension. The number of elements in each dimension of the array is determined by the number of elements at the corresponding nesting level in the array initializer. However, for each nested array initializer, the number of elements must be the same as the other array initializers at the same level.

Example: Create a two-dimensional array named - bbb - having the following integers arranged in rows and column shown below.

11	3
5	2
4	9
14	7

int[,] bbb = {{11, 3}, {5, 2}, {4, 9}, {14, 7}};

Note that in this example the number of elements in each dimension is not specified explicitly. The compiler computes it at compile time.

The number of elements specified in the array-initializer must always be a constant positive integer. It cannot be a variable. Moreover, the number of elements at each nesting level must match the corresponding dimension length. Thus, following array creation declarations are incorrect.

```
int i = 3;

int[] aaa = new int[i];
```

This declaration is incorrect because the number of elements specified (i) is not a constant. Note that the compiler must know the exact number of elements to be created at compile time. Since a variable (such as I) can take any value at run time, a variable cannot be employed as the specification of number of elements to create an array.

```
int[] ccc = new int[4] {10, 21, 2, 56, 77};
```

This declaration is incorrect because the number of elements specified (4) and the number of initial values in the list (5) do not match.

Accessing array elements

Once created the elements of an array can be accessed using an index. An index can be a constant or a variable of either int type or uint type or long type or ulong type. The index can also be an expression yeilding int, uint, long or ulong type value or values implicitly convertible into these type of values. The number of indices used must be equal to the dimension of the array being accessed.

The Array Class

An array is implemented as a class in System package of C#. The System.Array type is the base class from which all the array objects are derived. When you create an array using either array-initializer or using new method what is created is a subclass of System.Array base class. This class provides all the necessary functionalities of an array data type. A synopsis of some of the member methods of System.Array class is presented below.

System.Array.Reverse()

This method reverses the order of a given length of the given array. The syntax is given below.

System.Array.Clear()

This is a static method of the class. It clears the specified range of elements of the stored values. Numeric elements are assigned with 0 while reference type elements are assigned null. The sysntax of the method is given below.

System.Array.Clear(arrayname, lowerindex, higherindex);

Example: Clear all the elements having index values between 3 to 9 inclusive of the array names.

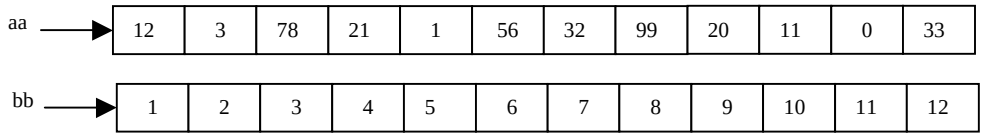
System.Array.Clear(names, 3, 9);

System.Array.Copy()

This method takes two array arguments and copies specified number of elements starting from specified index into another array from specified index position. The syntax is given below.

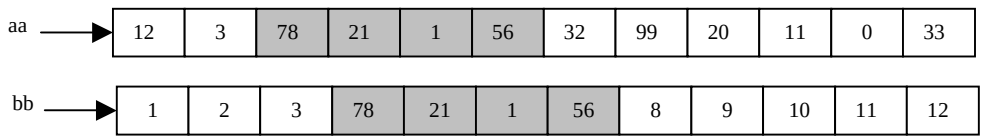
System.Array.Copy(SourceArr, SourceInd, DestArr,DestInd,Len);

Example: Copy 4 elements starting from third position (index 2) of array aa into array bb starting at fourth position (index 3) as shown below.



System.Array.Copy(aa, 2, bb, 3, 4);

The result is shown below.

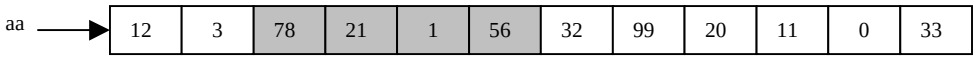


System.Array.Reverse()

This method reverses the order of a given length of the given array. The sysntax is given below.

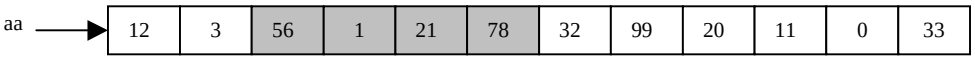
System.Array.Reverse(array, index, length);

Example: Reverse the order of 4 elements starting from third position (index 2) of the given array aa.



System.Array.Reverse(array, index, length);

The result is shown below.



Student Activity 1

1. What do you mean by "Array"? How it can be declared and initialized in a C# program?
2. Explain the different ways to initialize Arrays?
3. Name some of the methods available in System.Array class.
4. What are the differences between a fixed size array and a variable size array?
5. Write a program using a variable size integer array.
6. How can you create a three dimensional array?

3.3 STRUCTURES

A structure is a variable data type that contains a group of variables of different types including structure type and functions. In this sense structures are similar to classes. However, there is a significant difference between the two. Structures are value types while classes are reference types. Consequently, structures store data values directly and do not require heap allocation. On the other hand a class type variable contains a reference to the data stored elsewhere (called the object) on the heap memory.

Using structures a programmer can treat a group of related variables as a single unit. Thus, data items like complex numbers, coordinates of points etc. can be conveniently structured as structures. The primitive data types in C# such as int, double, and bool, are actually defined to be structures behind the scene.

Declaring structures

The declaration of a structure defines its template. It does not create any variable. After one has declared a structure one can create variables of that structure type just as any variable is created. A structure is declared as shown below.

```
struct-declaration:
attributes modifiers struct identifier1 interfaces body;
identifier2;
```

Where attributes is an optional set of attributes; modifiers is an optional set of modifiers; struct is mandatory keyword; identifier1 is the name of this structure type; interfaces is an optional list of interfaces that the structure may implement; body is mandatory part where the members of the structure are specified; and identifier2 is an optional comma separated variable names which if present are created as this structure type.

- A structure declaration may optionally include a sequence of structure modifiers. The modifiers may be one of the following.

  new

  public

protected

internal

private

These modifiers are common between classes and structures. Classes have a few more modifiers than listed above. The abstract and sealed modifiers are not applicable only in classes and not in structures.

- A structure declaration may implement one or more interfaces similar to classes.
- The body of a structure is enclosed within curly braces - { }. It is here that the members of the structures are specified.

  The members of a structure can a group of the following types.

  ```
  constant
  field
  method
  property
  event
  indexer
  operator
  constructor
  static-constructor
  type
  ```

Student Activity 2

1. What is a structure? How is it different from classes?
2. What are the different structure modifiers?

3.4 CLASS Vs STRUCTURE

Classes and structures appear to be identical. However, it is not quite so. They differ more than they are similar. Some of the most important differences are mentioned below.

Type

Structures are value types as against classes, which are reference types. A structure type variable directly contains the data of the structure. The same is not true with classes. A class type variable does not have data embedded with its name. It is just a reference (or pointer) that is stored at a location. The data associated with the class (object) is stored on the heap memory to which the name of the class points.

One of the consequences of this difference is that two variables of class type may reference the same object, and therefore it is possible for operations on one variable to affect the object referenced by the other variable. Structure variables do not reference each other. They each have their own copy of the data hence it is not possible for operations on one to affect the other.

It is because of this difference that structure type variables cannot be assigned null values while a class type variable may be assigned a null value.

For example, consider the program listed below.

```
using system;
class One
{
  struct Point
  {
    public int x, y, z;
```

```
        public Point(int x, int y, int z)
        {
           this.x = x;
           this.y = y;
           this.z = z;
        }
     }
     static void Main(string[] args)
     {
        Point a = new Point(100, 200, 300);
        Point b = a;
        Console.WriteLine("b.x before assignment " + b.x.ToString());
        a.x=400;
        Console.WriteLine("b.x after assignment " + b.x.ToString());
     }
  }
```

The output of the above program is shown below.

b.x before assignment 100

b.x before assignment 100

Note that the value stored in b.x remains unaffected even though the value of a.x is changed. This is because the assignment b = a creates a structure type (the type of a) variable b which has its own copy of x, y and z.

Inheritance

A class may be sub-classed to form a new class. The new class inherits properties and methods from the base-class. Even structure type variables implicitly inherit from class object. Unlike a class a structure type cannot specify a base class though it may implement interfaces. Structure types cannot be abstract and are always implicitly sealed. Therefore, abstract and sealed modifiers are not permitted in a structure declaration. Inheritance is not applicable to structures. Therefore, the members of a structure cannot have protected or protected internal accessibility. For the same reason the function members of a structure cannot be abstract or virtual.

Assignment

Assignment to a variable of a structure type creates a copy of the value being assigned. This differs from assignment to a variable of a class type, which copies the reference but not the object pointed by the reference.

Similar to assignment, a copy of the structure passed to or returned from a member function when the structure is passed as a value parameter. However, in order to pass a structure to a member function by reference a ref or out parameter must be used.

Default values

In C# the variables are initialized with default values at the time of their creation unlike C wherein the initial value could be garbage. The reference type variables, including a class variable, are initialized with null. Structures being value type are initialized differently. For a structure the default initialization takes place member-wise, i.e., each field is initialized with default values. For instance, numeric fields are initialized with 0.

Consider the Point structure once again.

```
struct Point
{
  public int x, y, z;
  public Point(int x, int y, int z)
  {
    this.x = x;
    this.y = y;
    this.z = z;
  }
}
```

If a new Point structure variable is created using new key word, the individual fields (i.e., x, y and z) are initialized with 0.

```
Point[] point1 = new Point[];
```

The value of each of the variables point1.x, point1.y and point1.z shall be set to 0.

Boxing and unboxing

While boxing and unboxing conversion of a class type variable the object remains the same only its reference changes to appropriate type. The same is not true with structures. Boxing and unboxing in this manner is not applicable to structures simply because structures are value types, not reference type. Therefore, in both the boxing and unboxing operations a copy of the structures being boxed or unboxed is created which does not reflect the changes done on the original structure variable.

This

The reserved word - this - is treated as a value when used within a constructor or member function of a class. Therefore, this can be used to refer to the instance for which the function member was invoked. However, it is not possible to assign a value to this in a member function of a class.

On the contrary, in case of a structure, this corresponds to an out parameter and within and member function of a structure, this corresponds to a ref parameter. In either case, this is treated as a variable. Therefore, it is possible to assign values to it and modify the entire structure for which the member function was invoked. It can also be passed as a ref or out parameter to a member function.

Field initializers

The fields of a structure get initialized to default values at the time of creation. Therefore, it is an error to initialize the fields with constants in the declaration. Consider the following code-snippet.

```
struct Point
{
  public int x = 100; // Error, initializer not permitted
  public int y = 200; // Error, initializer not permitted
  public int z = 300; // Error, initializer not permitted
}
```

The same can be achieved by introducing a constructor function to the structure and passing to it the initial values at the time of creation of the structure. See the following code fragment.

```
struct Point
{
  public int x, y, z;
  public Point(int x, int y, int z);
  {
    this.x = x;
    this.y = y;
    this.z = z;
  }
}
  Point point1 = new Point(100, 200, 300);
```

Note that field initialization is permitted for class variables (i.e., static variables). Thus the following code is correct.

```
struct Point
{
  public int x, y, z;
  static public NumberOfpoints = 0;  //no error!
  public Point(int x, int y, int z)
  {
    this.x = x;
    this.y = y;
    this.z = z;
  }
}
```

The Point class has been added with a static member - NumberOfPoints. This is a class variables and not instance variable, which is indicated by it being static. Therefore, it is no error to initialize with value.

Constructors

You have seen that a structure can have constructors and/or destructor just as a class has. However, unlike a class, a structure cannot declare a constructor having no parameters. In fact, every structure has a parameterless constructor implicitly provided which initializes the fields of the structure when created without calling its constructor. Besides, in case of a nested structure a constructor of the form base(…) is not allowed for the purpose of initialization as it happens in case of a class. A structure can also have static constructors which are similar to those of classes.

Destructors

While a class can have a explicit destructor member function a structure cannot have one.

Student Activity 3

1. What are the main differences between a classes and structures?
2. Explain boxing and unboxing conversions. How do they differ in classes and structures?
3. What is the difference between default and parameterized constructor?

3.5 SUMMARY

- An Array is a group of memory locations related by the fact that they all have the same name and same data type.
- An Array including more than one dimension is called a multidimensional array.
- In memory two-dimensional arrays are represent in two ways, as Row major form, Column major form.
- The System.Array class serves as the base class for all types of arrays. The System.Array is not itself an array type rather; it is a class type from which all array types are derived.
- A component of an array is referred to by its positions in the array whereas each component of a structure has a unique name.
- Structure members are stored in memory in the same order in which they are declared.
- The keywords struct can be used to declare a structure.
- In C# structure member can be accessed in the same way as class member access.
- A C# struct can also contain methods. The methods can be either static or non-static.
- In C# every value type implicitly has a public parameter less default constructor.
- There is no inheritance for structs as there is for classes. A struct cannot inherit form another struct or class and it cannot be the base class for a class.

3.6 KEYWORDS

Element: Each constituted value of an array is called an element.

Multi-dimensional Array: Arrays having more than two dimensions are called as Multi-dimensional arrays.

Structure: A structure is a collection of different data types in such a way that they can be referenced as a single unit.

Default Constructor : Parameter less constructor is called as default constructor.

Boxing: it is an implicit conversion of a value type to the type object.

Unboxing: It is an explicit conversion from the type object to a value type.

3.7 REVIEW QUESTIONS

1. Write a C# program to sort a two-dimensional array in order specified by user.
2. What are the differences between a fixed size array and a variable size array?
3. What will be the output of the following C# code?

```
using System;
public class ArrayTest
{
   public static void Main()
   {
      int d=0;
```

```
        int[][] myArray = new int[2][];
         myArray[0] = new int[5] {1,3,5,7,9};
         myArray[1] = new int[4] {2,4,6,8};
         for (int i=0; i < myArray.Length; i++)
            for (int j = 0 ; j < myArray[i].Length ; j++)
               d = d + myArray[i,j];
               Console.Write({0},d);
            }
      }
}
```

4. Write a program using a variable size integer array.
5. Write a program to accept 12 numbers in two dimensional arrays in row major form.
6. Write a program to demonstrate the use of different method of ArrayList Class.
7. What are the main differences between a class and a structure with methods?
8. What are the advantages of enumeration data type?
9. Explain the usages of boxing and unboxing conversions.
10. Dry-run the following program to find out its output.

```
using System;
public class En
{
   public enum MON: byte
   {January, February, November, December, March}
      public static void Main()
      {
         Array dayArray = Enum.GetValues(typeof(En.MON));
         foreach (MON m in dayArray)
         Console.WriteLine("Number {1} of En.MON is {0}", m,
         m.ToString("d"));
      }
}
```

3.8 FURTHER READINGS

CLR via C#, Second Edition (Paperback) by Jeffrey Richter, Publisher: Microsoft Press; 2nd edition (February 22, 2006)

Essential C# 2.0 (Microsoft .NET Development Series) (Paperback) by Mark Michaelis, Publisher: Addison-Wesley Professional (July 13, 2006)

C# Programming Language, The (2nd Edition) (Microsoft .NET Development Series) (Hardcover) by Anders Hejlsberg, Scott Wiltamuth, Peter Golde, Publisher: Addison-Wesley Professional; 2 edition (June 9, 2006)

Programming in the Key of C# (Paperback) by Charles Petzold, Publisher: Microsoft Press; 1 edition (July 30, 2003)

UNIT

4

CLASSES AND METHODS

L E A R N I N G O B J E C T I V E S

After studying this unit, you should be able to

- Define class and its type
- Know about class members
- Describe methods
- Describe override methods

U N I T S T R U C T U R E

4.1 INTRODUCTION

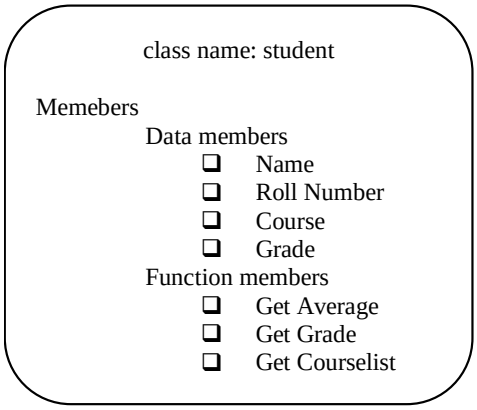
This chapter will introduce you to classes, which are the fundamental topic to be discussed in any Object-Oriented Programming (OOP) language. A class is a combination of related objects whereas each object is an instance or a copy of the corresponding class. Also we will deal with Methods and the various types of methods will be discussed in detail. Methods are blocks of code that perform some kind of action, or carry out functions such as printing, opening a dialog box, and so forth.

4.2 CLASS

Class is the central concept of Object Oriented Approach in programming. In most of the object-oriented programming languages classes are implemented as a structure having additional features. Class implements the Object Oriented Approach in programming. A class is a data structure that may contain the following elements:

- data members such as constants, variables and events
- function members such as methods, properties, indexers, operators, constructors, and destructors
- nested types.

Class types support inheritance, a mechanism whereby a derived class can extend and specialize a base class. Consider the pictorial representation of a class named student.



A class-declaration consists of an optional set of attributes, followed by an optional set of class-modifiers , followed by the keyword class and an identifier that names the class, followed by an optional class-base specification, followed by a class-body, optionally followed by a semicolon.

A class-declaration syntax is given below.

```
class-modifiers class-name class-body
```

A class-modifier may be one or more of the following:

```
new
public
protected
internal
private
abstract
sealed
```

The keyword new before a class name creates an object of the specified class type. If a class is declared as public, other classes can access it. Rest of the modifiers will be dealt with in the subsequent sections.

Class-name is any valid identifier that uniquely identifies each class. The class-body is gropu of statements enclosed within curly braces - { and }.

Thus, the student class may be written as:

```
Public class Student
{
     string name;
     int rollNo;
     string course;
     char grade;
     char GetGrade();
     float GetAverage();
     string[] GetCourseList();
}
```

Note that it is an error for the same modifier to appear multiple times in a class declaration. The new modifier is only permitted on nested classes. It specifies that the class hides an inherited member by the same name. The public, protected, internal, and private modifiers control the accessibility of the class. Depending on the context in which the class declaration occurs, some of these modifiers may not be permitted.

Abstract classes

The abstract modifier is used to indicate that a class is incomplete and intended only to be a base class of other classes. An abstract class differs from a non-abstract class is the following ways:

- An abstract class cannot be instantiated directly, and it is an error to use the new operator on an abstract class. While it is possible to have variables and values whose compile-time types are abstract, such variables and values will necessarily either be null or contain references to instances of non-abstract classes derived from the abstract types.
- An abstract class is permitted (but not required) to contain abstract members.
- An abstract class cannot be sealed.

When a non-abstract class is derived from an abstract class, the non-abstract class must include actual implementations of all inherited abstract members. Such implementations are provided by overriding the abstract members (see the following code).

```
abstract class AbstractOne
{
  public abstract void FunctionOne();
}
abstract class AbstractTwo:AbstractOne
{
  public void FunctionTwo() {}
}
class AbstractThree : AbstractTwo
{
  public override void FunctionOne()
  {
    // actual implementation of FunctionOne
  }
}
```

Here, the abstract class AbstractOne introduces an abstract method FunctionOne. Class AbstractTwo introduces an additional method FunctionTwo, but doesn't provide an implementation. AbstractTwo must therefore also be declared abstract. Class AbstractThree overrides FunctionOne and provides an actual implementation. Since there are no outstanding abstract members in AbstractThree, it is permitted (but not required) to be non-abstract.

Sealed classes

The sealed modifier is used to prevent derivation from a class. An error occurs if a sealed class is specified as the base class of another class. A sealed class cannot be an abstract class.

The sealed modifier is primarily used to prevent unintended derivation, but it also enables certain run-time optimizations. In particular, because a sealed class is known to never have any derived classes, it is possible to transform virtual function member invocations on sealed class instances into non-virtual invocations.

Class base specification

A class declaration may include a class-base specification which defines the direct base class of the class and the interfaces implemented by the class.

For instance, consider the following code.

```
Class DerivedClass : BaseClass, InterfaceName
{}
```

Base classes

When a class-type is included in the class-base, it specifies the direct base class of the class being declared. If a class declaration has no class-base, or if the class-base lists only interface types, the direct base class is assumed to be object. A class inherits members from its direct base class.

In the example code given below,

```
class One
{}
class Two:One
{}
```

class One is said to be the direct base class of Two, and Two is said to be derived from One. Since One does not explicitly specify a direct base class, its direct base class is implicitly System.object.

The direct base class of a class type must be at least as accessible as the class type itself. For example, it is an error for a public class to derive from a private or internal class.

The direct base class of a class type must not be any of the following types: System.Array, System.Delegate, System.Enum, or System.ValueType.

The base classes of a class are the direct base class and its base classes. In other words, the set of base classes is the transitive closure of the direct base class relationship. Referring to the example above, the base classes of B are A and object.

Except for class object, every class has exactly one direct base class. The object class has no direct base class and is the ultimate base class of all other classes.

When a class B derives from a class A, it is an error for A to depend on B. A class directly depends on its direct base class (if any) and directly depends on the class within which it is immediately nested (if any). Given this definition, the complete set of classes upon which a class depends is the transitive closure of the directly depends on relationship (see the example code given below).

```
class One: Two {}
class Two: Three {}
class Three: One {}
```

This is in error because the classes circularly depend on themselves. Likewise, the example,

```
class One: Two.Three {}
class Two: One
{
  public class Three {}
}
```

is in error because One depends on Two.Three (its direct base class), which depends on Two (its immediately enclosing class), which circularly depends on One. Note that a class does not depend on the classes that are nested within it. Consider the following code.

```
class One
{
  class Two: One {}
}
```

Class Two depends on One (because One is both its direct base class and its immediately enclosing class), but One does not depend on Two (since Two is neither a base class nor an enclosing class of One). Thus, the example is valid.

It is not possible to derive from a sealed class. Consider the code listed below.

```
sealed class One {}
class Two: One {}
```

This is an error. Class Two attempts to derive from the sealed class One which is not allowed.

A class-base specification may include a list of interface types, in which case the class is said to implement the given interface types.

4.3 CLASS MEMBERS

The class-body of a class defines the members of the class. The members of a class consist of the members introduced by its class-member-declarations and the members inherited from the direct base class. The members of a class are divided into the following categories:

- Constants, which represent constant values associated with the class.
- Fields, which are the variables of the class.
- Methods, which implement the computations and actions that can be performed by the class.
- Properties, which define named attributes and the actions associated with reading and writing those attributes.
- Events, which define notifications that are generated by the class.
- Indexers, which permit instances of the class to be indexed in the same way as arrays.
- Operators, which define the expression operators that can be applied to instances of the class.
- Instance constructors, which implement the actions required to initialize instances of the class
- Destructors, which implement the actions to perform before instances of the class are permanently discarded.
- Static constructors, which implement the actions required to initialize the class itself.
- Types, which represent the types that are local to the class.

Members that contain executable code are collectively known as the function members of the class. The function members of a class are the methods, properties, indexers, operators, constructors, and destructors of the class.

A class-declaration creates a new declaration space, and the class-member-declarations immediately contained by the class-declaration introduce new members into this declaration space. The following rules apply to class-member-declarations:

- Constructors and destructors must have the same name as the immediately enclosing class. All other members must have names that differ from the name of the immediately enclosing class.
- The name of a constant, field, property, event, or type must differ from the names of all other members declared in the same class.
- The name of a method must differ from the names of all other non-methods declared in the same class. In addition, the signature of a method must differ from the signatures of all other methods declared in the same class.
- The signature of an constructor must differ from the signatures of all other constructors declared in the same class.
- The signature of an indexer must differ from the signatures of all other indexers declared in the same class.
- The signature of an operator must differ from the signatures of all other operators declared in the same class.

The inherited members of a class are specifically not part of the declaration space of a class. Thus, a derived class is allowed to declare a member with the same name or signature as an inherited member (which in effect hides the inherited member).

The new modifier

A class-member-declaration is permitted to declare a member with the same name or signature as an inherited member. When this occurs, the derived class member is said to hide the base class member. Hiding an inherited member is not considered an error, but it does cause the compiler to issue a warning. To suppress the warning, the declaration of the derived class member can include a new modifier to indicate that the derived member is intended to hide the base member.

If a new modifier is included in a declaration that doesn't hide an inherited member, a warning is issued to that effect. This warning is suppressed by removing the new modifier.

It is an error to use the new and override modifiers in the same declaration.

Access modifiers

A class-member-declaration can have any one of the five possible types of declared accessibility :public, protected internal, protected, internal, or private. Except for the protected internal combination, it is an error to specify more than one access modifier. When a class-member-declaration does not include any access modifiers, the declaration defaults to private declared accessibility.

Constituent types

Types that are referenced in the declaration of a member are called the constituent types of the member. Possible constituent types are the type of a constant, field, property, event, or indexer, the return type of a method or operator, and the parameter types of a method, indexer, operator, or constructor. The constituent types of a member must be at least as accessible as the member itself.

Static and instance members

Members of a class are either static members or instance members. Generally speaking, it is useful to think of static members as belonging to classes and instance members as belonging to objects (instances of classes).

When a field, method, property, event, operator, or constructor declaration includes a static modifier, it declares a static member. In addition, a constant or type declaration implicitly declares a static member. Static members have the following characteristics:

- When a static member is referenced in a member-access of the form E.M, E must denote a type. It is an error for E to denote an instance.
- A static field identifies exactly one storage location. No matter how many instances of a class are created, there is only ever one copy of a static field.
- A static function member (method, property, indexer, operator, or constructor) does not operate on a specific instance, and it is an error to refer to this in a static function member.

When a field, method, property, event, indexer, constructor, or destructor declaration does not include a static modifier, it declares an instance member. An instance member is sometimes called a non-static member. Instance members have the following characteristics:

- When an instance member is referenced in a member-access of the form E.M, E must denote an instance. It is an error for E to denote a type.

  very instance of a class contains a separate copy of all instance fields of the class.
- n instance function member (method, property accessor, indexer accessor, constructor, or destructor) operates on a given instance of the class, and this instance can be accessed as this.

The following example illustrates the rules for accessing static and instance members:

```
class TestClass
{
  int x;
  static int y;
  void FunctionOne()
  {
    x = 1;      // Ok, same as this.x = 1
    y = 1;      // Ok, same as Test.y = 1
  }
```

```
    static void FunctionTwo()
    {
      x = 1;      // Error, cannot access this.x
      y = 1;      // Ok, same as Test.y = 1
    }
    static void Main()
    {
      TestClass t = new TestClass();
      t.x = 1;      // Ok
      t.y = 1;      // Error, cannot access static member
                    // through instance
    Test.x = 1;     // Error, cannot access instance member
                    // through type
    Test.y = 1;     // Ok
    }
}
```

The FunctionOne method shows that in an instance function member, a simple-name can be used to access both instance members and static members. The FunctionTwo method shows that in a static function member, it is an error to access an instance member through a simple-name. The Main method shows that in a member-access, instance members must be accessed through instances, and static members must be accessed through types.

Nested types

Constants

A constant is a class member that represents a constant value. A constant type is a value that can be computed at compile-time. A constant-declaration introduces one or more constants of a given type.

A constant-declaration may include a set of attributes, a new modifier, and a valid combination of the four access modifiers. The attributes and modifiers apply to all of the members declared by the constant-declaration. Even though constants are considered static members, a constant-declaration neither requires nor allows a static modifier.

The type of a constant-declaration specifies the type of the members introduced by the declaration. The type is followed by a list of constant-declarators, each of which introduces a new member. A constant-declarator consists of an identifier that names the member, followed by an "=" token, followed by a constant-expression that gives the value of the member.

The type specified in a constant declaration must be sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, bool, string, an enum-type, or a reference-type. Each constant-expression must yield a value of the target type or of a type that can be converted to the target type by an implicit conversion.

The type of a constant must be at least as accessible as the constant itself.

A constant can itself participate in a constant-expression. Thus, a constant may be used in any construct that requires a constant-expression. Examples of such constructs include case labels, goto case statements, enum member declarations, attributes, and other constant declarations.

A constant-expression is an expression that can be fully evaluated at compile-time. Since the only way to create a non-null value of a reference-type other than string is to apply the new operator, and since the new operator is not permitted in a constant-expression, the only possible value for constants of reference-types other than string is null.

When a symbolic name for a constant value is desired, but when the type of the value is not permitted in a constant declaration or when the value cannot be computed at compile-time by a constant-expression, a readonly field may be used instead.

A constant declaration that declares multiple constants is equivalent to multiple declarations of single constants with the same attributes, modifiers, and type. For example

```
  class One
  {
    public const double X = 1.0, Y = 2.0, Z = 3.0;
  }
is equivalent to
  class One
  {
    public const double X = 1.0;
    public const double Y = 2.0;
    public const double Z = 3.0;
  }
```

Constants are permitted to depend on other constants within the same program as long as the dependencies are not of a circular nature. The compiler automatically arranges to evaluate the constant declarations in the appropriate order. Consider the code listed below.

```
class One
  {
    public const int X = Two.Z + 1;
    public const int Y = 10;
  }
  class Two
  {
    public const int Z = A.Y + 1;
  }
```

The compiler first evaluates Y, then evaluates Z, and finally evaluates X, producing the values 10, 11, and 12. Constant declarations may depend on constants from other programs, but such dependencies are only possible in one direction. Referring to the example above, if One and Two were declared in separate programs, it would be possible for One.X to depend on Two.Z, but Two.Z could then not simultaneously depend on One.Y.

Fields

A field is a member that represents a variable associated with an object or class. A field-declaration introduces one or more fields of a given type.

A field-declaration may include a set of attributes, a new modifier, a valid combination of the four access modifiers, a static modifier, and a readonly modifier. The attributes and modifiers apply to all of the members declared by the field-declaration.

The type of a field-declaration specifies the type of the members introduced by the declaration. The type is followed by a list of variable-declarators, each of which introduces a new member. A variable-declarator consists of an identifier that names the member, optionally followed by an "=" token and a variable-initializer. that gives the initial value of the member.

The type of a field must be at least as accessible as the field itself.

The value of a field is obtained in an expression using a simple-name or a member-access. The value of a field is modified using an assignment. The value of a field can be both obtained and modified using postfix increment and decrement operators and prefix increment and decrement operators.

A field declaration that declares multiple fields is equivalent to multiple declarations of single fields with the same attributes, modifiers, and type. For example

```
  class One
  {
    public static int X = 1, Y, Z = 100;
  }
is equivalent to
  class One
  {
    public static int X = 1;
    public static int Y;
    public static int Z = 100;
  }
```

Static and instance fields

When a field-declaration includes a static modifier, the fields introduced by the declaration are static fields. When no static modifier is present, the fields introduced by the declaration are instance fields. Static fields and instance fields are two of the several kinds of variables supported by C#, and are at times referred to as static variables and instance variables.

A static field identifies exactly one storage location. No matter how many instances of a class are created, there is only ever one copy of a static field. A static field comes into existence when the type in which it is declared is loaded, and ceases to exist when the type in which it is declared is unloaded.

Every instance of a class contains a separate copy of all instance fields of the class. An instance field comes into existence when a new instance of its class is created, and ceases to exist when there are no references to that instance and the destructor of the instance has executed.

When a field is referenced in a member-access of the form E.M, if M is a static field, E must denote a type, and if M is an instance field, E must denote an instance.

Readonly fields

When a field-declaration includes a readonly modifier, assignments to the fields introduced by the declaration can only occur as part of the declaration or in a constructor in the same class. Specifically, assignments to a readonly field are permitted only in the following contexts:

- n the variable-declarator that introduces the field (by including a variable-initializer in the declaration).
- or an instance field, in the instance constructors of the class that contains the field declaration, or for a static field, in the static constructor of the class the that contains the field declaration. These are also the only contexts in which it is valid to pass a readonly field as an out or ref parameter.

Attempting to assign to a readonly field or pass it as an out or ref parameter in any other context is an error.

A static readonly field is useful when a symbolic name for a constant value is desired, but when the type of the value is not permitted in a const declaration or when the value cannot be computed at compile-time. Consider the following code.

```
  public class Color
  {
    public static readonly Color Black = new Color(0, 0, 0);
    public static readonly Color White = new Color(255, 255, 255);
    public static readonly Color Red = new Color(255, 0, 0);
    public static readonly Color Green = new Color(0, 255, 0);
    public static readonly Color Blue = new Color(0, 0, 255);
    private byte red, green, blue;
```

```
public Color(byte r, byte g, byte b)
{
  red = r;
  green = g;
  blue = b;
}
}
```

The Black, White, Red, Green, and Blue members cannot be declared as const members because their values cannot be computed at compile-time. However, declaring the members as static readonly fields has much the same effect.

Field initialization

The initial value of a field is the default value of the field's type. When a class is loaded, all static fields are initialized to their default values, and when an instance of a class is created, all instance fields are initialized to their default values. It is not possible to observe the value of a field before this default initialization has occurred, and a field is thus never "uninitialized". Consider the following code.

```
class TestClass
{
  static bool b;
  int i;
  static void Main()
  {
    TestClass t = new TestClass();
    Console.WriteLine("b = {0}, i = {1}", b, t.i);
  }
}
```

This produces the output

```
b = False, i = 0
```

because b is automatically initialized to its default value when the class is loaded and i is automatically initialized to its default value when an instance of the class is created.

Variable initializers

Field declarations may include variable-initializers. For static fields, variable initializers correspond to assignment statements that are executed when the class is loaded. For instance fields, variable initializers correspond to assignment statements that are executed when an instance of the class is created.

Consider the following code.

```
class TestClass
{
    static double x = Math.Sqrt(2.0);
    int i = 100;
    string s = "Hello";
    static void Main() {
    TestClass a = new TestClass();
    Console.WriteLine("x = {0}, i = {1}, s = {2}", x, a.i, a.s);
  }
}
```

It produces the output

```
x = 1.414213562373095, i = 100, s = Hello
```

because an assignment to x occurs when the class is loaded and assignments to i and s occur when an new instance of the class is created.

The default value initialization occurs for all fields, including fields that have variable initializers. Thus, when a class is loaded, all static fields are first initialized to their default values, and then the static field initializers are executed in textual order. Likewise, when an instance of a class is created, all instance fields are first initialized to their default values, and then the instance field initializers are executed in textual order.

It is possible for static fields with variable initializers to be observed in their default value state, though this is strongly discouraged as a matter of style (see the following code).

```
class TestClass
{
   static int a = b + 1;
   static int b = a + 1;
   static void Main() {
   Console.WriteLine("a = {0}, b = {1}", a, b);
   }
}
```

This program exhibits this behavior. Despite the circular definitions of a and b, the program is legal. It produces the output

```
a = 1, b = 2
```

because the static fields a and b are initialized to 0 (the default value for int) before their initializers are executed. When the initializer for a runs, the value of b is zero, and so a is initialized to 1. When the initializer for b runs, the value of a is already 1, and so b is initialized to 2.

Student Activity 1

1. What are abstract classes?
2. Define a class student with studno, studname, class and grade.
3. Define sealed class.
4. Name different types of members we can put in a class.
5. What do you understand by static and instance members?
6. What is a field? What do you understand by static and instance fields?

4.4 METHODS

A method is a member that implements a computation or action that can be performed by an object or class. Methods are declared using method-declarations. A method-declaration may include a set of attributes, a new modifier, an extern modifier, a valid combination of the four access modifiers, and a valid combination of the static, virtual, override, and abstract modifiers. In addition, a method that includes the override modifier may also include the sealed modifier.

The static, virtual, override, and abstract modifiers are mutually exclusive except in one case. The abstract and override modifiers may be used together so that an abstract method can override a virtual one.

The return-type of a method declaration specifies the type of the value computed and returned by the method. The return-type is void if the method does not return a value.

The member-name specifies the name of the method. Unless the method is an explicit interface member implementation, the member-name is simply an identifier. For an explicit interface member implementation, the member-name consists of an interface-type followed by a "." and an identifier. The optional formal-parameter-list specifies the parameters of the method.

The return-type and each of the types referenced in the formal-parameter-list of a method must be at least as accessible as the method itself. For abstract and extern methods, the method-body consists simply of a semicolon. For all other methods, the method-body consists of a block which specifies the statements to execute when the method is invoked.

The name and the formal parameter list of a method defines the signature of the method. Specifically, the signature of a method consists of its name and the number, modifiers, and types of its formal parameters. The return type is not part of a method's signature, nor are the names of the formal parameters.

The name of a method must differ from the names of all other non-methods declared in the same class. In addition, the signature of a method must differ from the signatures of all other methods declared in the same class.

Method parameters

The parameters of a method, if any, are declared by the method's formal-parameter-list. The formal parameter list consists of one or more fixed-parameters optionally followed by a single parameter-array, all separated by commas.

A fixed-parameter consists of an optional set of attributes, an optional ref or out modifier, a type, and an identifier. Each fixed-parameter declares a parameter of the given type with the given name. A parameter-array consists of an optional set of attributes, a params modifier, an array-type, and an identifier. A parameter array declares a single parameter of the given array type with the given name.

The array-type of a parameter array must be a single-dimensional array type. In a method invocation, a parameter array permits either a single argument of the given array type to be specified, or it permits zero or more arguments of the array element type to be specified. A method declaration creates a separate declaration space for parameters and local variables. Names are introduced into this declaration space by the formal parameter list of the method and by local variable declarations in the block of the method. All names in the declaration space of a method must be unique. Thus, it is an error for a parameter or local variable to have the same name as another parameter or local variable.

A method invocation creates a copy, specific to that invocation, of the formal parameters and local variables of the method, and the argument list of the invocation assigns values or variable references to the newly created formal parameters. Within the block of a method, formal parameters can be referenced by their identifiers in simple-name expressions.

There are four kinds of formal parameters:

- Value parameters, which are declared without any modifiers.
- Reference parameters, which are declared with the ref modifier.
- Output parameters, which are declared with the out modifier.
- Parameter arrays, which are declared with the params modifier.
- The ref and out modifiers are part of a method's signature, but the params modifier is not.

Value parameters

A parameter declared with no modifiers is a value parameter. A value parameter corresponds to a local variable that gets its initial value from the corresponding argument supplied in the method invocation.

When a formal parameter is a value parameter, the corresponding argument in a method invocation must be an expression of a type that is implicitly convertible to the formal parameter type.

A method is permitted to assign new values to a value parameter. Such assignments only affect the local storage location represented by the value parameter-they have no effect on the actual argument given in the method invocation.

Reference parameters

A parameter declared with a ref modifier is a reference parameter. Unlike a value parameter, a reference parameter does not create a new storage location. Instead, a reference parameter represents the same storage location as the variable given as the argument in the method invocation.

When a formal parameter is a reference parameter, the corresponding argument in a method invocation must consist of the keyword ref followed by a variable-reference of the same type as the formal parameter. A variable must be definitely assigned before it can be passed as a reference parameter.

Within a method, a reference parameter is always considered definitely assigned. Consider the following example.

```
class TestClass
{
  static void Swap(ref int x, ref int y) {
  int temp = x;
  x = y;
  y = temp;
}
static void Main() {
  int i = 1, j = 2;
  Swap(ref i, ref j);
  Console.WriteLine("i = {0}, j = {1}", i, j);
}
}
```

It produces the output

```
i = 2, j = 1
```

For the invocation of Swap in Main, x represents i and y represents j. Thus, the invocation has the effect of swapping the values of i and j.

In a method that takes reference parameters it is possible for multiple names to represent the same storage location. In the following example,

```
class One
{
  string s;
  void FunctionOne(ref string a, ref string b)
  {
    s = "One";
    a = "Two";
    b = "Three";
  }
  void FunctionTwo()
  {
    FunctionOne(ref s, ref s);
  }
}
```

the invocation of FunctionOne in FunctionTwo passes a reference to s for both a and b. Thus, for that invocation, the names s, a, and b all refer to the same storage location, and the three assignments all modify the instance field s.

Output parameters

A parameter declared with an out modifier is an output parameter. Similar to a reference parameter, an output parameter does not create a new storage location. Instead, an output parameter represents the same storage location as the variable given as the argument in the method invocation.

When a formal parameter is an output parameter, the corresponding argument in a method invocation must consist of the keyword out followed by a variable-reference of the same type as the formal parameter. A variable need not be definitely assigned before it can be passed as an output parameter, but following an invocation where a variable was passed as an output parameter, the variable is considered definitely assigned.

Within a method, just like a local variable, an output parameter is initially considered unassigned and must be definitely assigned before its value is used.

Every output parameter of a method must be definitely assigned before the method returns. Output parameters are typically used in methods that produce multiple return values. Consider the following code.

```
class TestClass
{
   static void SplitPath(string path, out string dir, out string name)
   {
      int i = path.Length;
      while (i > 0)
      {
         char ch = path[i - 1];
         if (ch == '\\' || ch == '/' || ch == ':')
         break;
         i--;
      }
      dir = path.Substring(0, i);
      name = path.Substring(i);
   }
static void Main()
{
   string dir, name;
   SplitPath("c:\\Windows\\System\\hello.txt", out dir, out name);
   Console.WriteLine(dir);
   Console.WriteLine(name);
}
}
```

The example produces the output:

```
c:\Windows\System\
hello.txt
```

Note that the dir and name variables can be unassigned before they are passed to SplitPath, and that they are considered definitely assigned following the call.

Parameter arrays

A parameter declared with a params modifier is a parameter array. If a formal parameter list includes a parameter array, it must be the last parameter in the list and it must be of a single-dimensional array type. For example, the types string[ ] and string[ ][ ] can be used as the type of a parameter array, but the type string[,] can not. It is not possible to combine the params modifier with the ref and out modifiers.

A parameter array permits arguments to be specified in one of two ways in a method invocation:

- The argument given for a parameter array can be a single expression of a type that is implicitly convertible to the parameter array type. In this case, the parameter array acts precisely like a value parameter.
- Alternatively, the invocation can specify zero or more arguments for the parameter array, where each argument is an expression of a type that is implicitly convertible to the element type of the parameter array. In this case, the invocation creates an instance of the parameter

array type with a length corresponding to the number of arguments, initializes the elements of the array instance with the given argument values, and uses the newly created array instance as the actual argument.

Except for allowing a variable number of arguments in an invocation, a parameter array is precisely equivalent to a value parameter of the same type. Consider the following code.

```
class TestClass
{
  static void FunctionOne(params int[] args)
  {
    Console.WriteLine("Array contains {0} elements:", args.Length);
    foreach (int i in args) Console.Write(" {0}", i);
    Console.WriteLine();
  }
  static void Main()
  {
    int[] a = {1, 2, 3};
    FunctionOne(a);
    FunctionOne(10, 20, 30, 40);
    FunctionOne();
  }
}
```

It produces the output

```
Array contains 3 elements: 1 2 3
Array contains 4 elements: 10 20 30 40
Array contains 0 elements:
```

The first invocation of FunctionOne simply passes the array a as a value parameter. The second invocation of FunctionOne automatically creates a four-element int[ ] with the given element values and passes that array instance as a value parameter. Likewise, the third invocation of FunctionOne creates a zero-element int[ ] and passes that instance as a value parameter. The second and third invocations are precisely equivalent to writing:

```
FunctionOne(new int[] {10, 20, 30, 40});
FunctionOne(new int[] {});
```

When performing overload resolution, a method with a parameter array may be applicable either in its normal form or in its expanded form. The expanded form of a method is available only if the normal form of the method is not applicable and only if a method with the same signature as the expanded form is not already declared in the same type. Consider the following code.

```
class TestClass
{
  static void FunctionOne(params object[] a)
  {
    Console.WriteLine("FunctionOne(object[])");
  }
  static void FunctionOne()
  {
    Console.WriteLine("FunctionOne()");
  }
  static void FunctionOne(object a0, object a1)
  {
    Console.WriteLine("FunctionOne(object,object)");
  }
```

```
    static void Main()
    {
      FunctionOne();
      FunctionOne(1);
      FunctionOne(1, 2);
      FunctionOne(1, 2, 3);
      FunctionOne(1, 2, 3, 4);
    }
  }
```

It produces the output

```
FunctionOne();
FunctionOne(object[]);
FunctionOne(object,object);
FunctionOne(object[]);
FunctionOne(object[]);
```

In the example, two of the possible expanded forms of the method with a parameter array are already included in the class as regular methods. These expanded forms are therefore not considered when performing overload resolution, and the first and third method invocations thus select the regular methods. When a class declares a method with a parameter array, it is not uncommon to also include some of the expanded forms as regular methods. By doing so it is possible to avoid the allocation of an array instance that occurs when an expanded form of a method with a parameter array is invoked.

When the type of a parameter array is object[ ], a potential ambiguity arises between the normal form of the method and the expended form for a single object parameter. The reason for the ambiguity is that an object[ ] is itself implicitly convertible to type object. The ambiguity presents no problem, however, since it can be resolved by inserting a cast if needed. Consider the following code.

```
  class TestClass
  {
    static void FunctionOne(params object[] args)
    {
      foreach (object o in a)
      {
        Console.Write(o.GetType().FullName);
        Console.Write(" ");
      }
      Console.WriteLine();
    }
    static void Main()
    {
      object[] a = {1, "Hello", 123.456};
      object o = a;
      FunctionOne(a);
      FunctionOne((object)a);
      FunctionOne(o);
      FunctioOne((object[])o);
    }
  }
```

It produces the following output.

```
System.Int32 System.String System.Double
System.Object[]
System.Object[]
System.Int32 System.String System.Double
```

In the first and last invocations of FunctionOne, the normal form of FunctionOne is applicable because an implicit conversion exists from the argument type to the parameter type (both are of type object[ ]). Thus, overload resolution selects the normal form of FunctionOne, and the argument is passed as a regular value parameter.

In the second and third invocations, the normal form of FunctionOne is not applicable because no implicit conversion exists from the argument type to the parameter type (type object cannot be implicitly converted to type object[ ]). However, the expanded form of FunctionOne is applicable, and it is therefore selected by overload resolution. As a result, a one-element object[ ] is created by the invocation, and the single element of the array is initialized with the given argument value (which itself is a reference to an object[ ]).

Static and instance methods

When a method declaration includes a static modifier, the method is said to be a static method. When no static modifier is present, the method is said to be an instance method.

A static method does not operate on a specific instance, and it is an error to refer to this in a static method. It is furthermore an error to include a virtual, abstract, or override modifier on a static method.

An instance method operates on a given instance of a class, and this instance can be accessed as this.

Virtual methods

When an instance method declaration includes a virtual modifier, the method is said to be a virtual method. When no virtual modifier is present, the method is said to be a non-virtual method.

It is an error for a method declaration that includes the virtual modifier to also include any one of the static, abstract, or override modifiers.

The implementation of a non-virtual method is invariant: The implementation is the same whether the method is invoked on an instance of the class in which it is declared or an instance of a derived class. In contrast, the implementation of a virtual method can be changed by derived classes. The process of changing the implementation of an inherited virtual method is known as overriding the method.

In a virtual method invocation, the run-time type of the instance for which the invocation takes place determines the actual method implementation to invoke. In a non-virtual method invocation, the compile-time type of the instance is the determining factor. In precise terms, when a method named N is invoked with an argument list A on an instance with a compile-time type C and a run-time type R (where R is either C or a class derived from C), the invocation is processed as follows:

- First, overload resolution is applied to C, N, and A, to select a specific method M from the set of methods declared in and inherited by C.
- Then, if M is a non-virtual method, M is invoked.
- Otherwise, M is a virtual method, and the most derived implementation of M with respect to R is invoked.

For every virtual method declared in or inherited by a class, there exists a most derived implementation of the method with respect to that class. The most derived implementation of a virtual method M with respect to a class R is determined as follows:

- If R contains the introducing virtual declaration of M, then this is the most derived implementation of M.
- Otherwise, if R contains an override of M, then this is the most derived implementation of M.
- Otherwise, the most derived implementation of M is the same as that of the direct base class of R.

The following example illustrates the differences between virtual and non-virtual methods:

```
class One
{
  public void FunctionOne()
  {
    Console.WriteLine("One. FunctionOne");
  }
  public virtual void FunctionTwo()
  {
    Console.WriteLine("One. FunctionTwo");
  }
}
class Two: One
{
  new public void FunctionOne()
{
  Console.WriteLine("Two. FunctionOne");
}
public override void FunctionTwo()
{
Console.WriteLine("Two. FunctionTwo");
}
}
class TestClass
{
  static void Main()
{
    Two b = new Two();
    One a = b;
    a.FunctionOne();
    b.FunctionOne();
    a.FunctionTwo();
    b.FunctionTwo();
  }
}
```

In the example, One introduces a non-virtual method FunctionOne and a virtual method FunctionTwo. The class FunctionTwo introduces a new non-virtual method FunctionOne, thus hiding the inherited FunctionOne, and also overrides the inherited method FunctionTwo. The example produces the output:

```
One.FunctionOne
Two.FunctionOne
Two.FunctionTwo
Two.FunctionTwo
```

Notice that the statement One.FunctionTwo() invokes Two.FunctionTwo, not one.FunctionTwo. This is because the run-time type of the instance (which is Two), not the compile-time type of the instance (which is One), determines the actual method implementation to invoke.

Because methods are allowed to hide inherited methods, it is possible for a class to contain several virtual methods with the same signature. This does not present an ambiguity problem, since all but the most derived method are hidden. Consider the following code.

```
class One
{
  public virtual void FunctionOne()
```

```
{
  Console.WriteLine("One.FunctionOne");
}
}
class Two: One
{
  public override void FunctionOne()
  {
  Console.WriteLine("Two.FunctionOne");
  }
}
class Three: Two
{
  new public virtual void FunctionOne()
  {
    Console.WriteLine("Three.FunctionONe");
  }
}
class Four: Three
{
  public override void FunctionOne()
  {
    Console.WriteLine("Four.FunctionOne");
  }
}
class TestClass
{
  static void Main()
  {
    Four d = new Four();
    One a = d;
    Two b = d;
    Three c = d;
    a.FunctionOne();
    b.FunctionOne();
    c.FunctionOne();
    d.FunctionOne();
  }
}
```

Classes Three and Four classes contain two virtual methods with the same signature: The one introduced by One and the one introduced by Three. The method introduced by Three hides the method inherited from One. Thus, the override declaration in Four overrides the method introduced by Three, and it is not possible for Four to override the method introduced by One. The example produces the output:

```
Two.FunctionOne
Two.FunctionOne
Four.FunctionOne
Four.FunctionOne
```

Note that it is possible to invoke the hidden virtual method by accessing an instance of D through a less derived type in which the method is not hidden.

Override methods

When an instance method declaration includes an override modifier, the method is said to be an override method. An override method overrides an inherited virtual method with the same signature. Whereas a virtual method declaration introduces a new method, an override method declaration specializes an existing inherited virtual method by providing a new implementation of the method.

It is an error for an override method declaration to include any one of the new, static, or virtual modifiers. An override method declaration may include the abstract modifier. This enables a virtual method to be overridden by an abstract method.

The method overridden by an override declaration is known as the overridden base method. For an override method M declared in a class C, the overridden base method is determined by examining each base class of C, starting with the direct base class of C and continuing with each successive direct base class, until an accessible method with the same signature as M is located. For purposes of locating the overridden base method, a method is considered accessible if it is public, if it is protected, if it is protected internal, or if it is internal and declared in the same program as C.

A compile-time error occurs unless all of the following are true for an override declaration:

- An overridden base method can be located as described above.
- The overridden base method is a virtual, abstract, or override method. In other words, the overridden base method cannot be static or non-virtual.
- The overridden base method is not a sealed method.
- The override declaration and the overridden base method have the same declared accessibility. In other words, an override declaration cannot change the accessibility of the virtual method.

An override declaration can access the overridden base method using a base-access. Consider the following code.

```
class One
{
   int x;
   public virtual void PrintFields()
   {
      Console.WriteLine("x = {0}", x);
   }
}
class Two: One
{
   int y;
   public override void PrintFields()
   {
      base.PrintFields();
      Console.WriteLine("y = {0}", y);
   }
}
```

Here the base.PrintFields() invocation in Two invokes the PrintFields method declared in One. A base-access disables the virtual invocation mechanism and simply treats the base method as a non-virtual method. Had the invocation in Two been written ((One)this).PrintFields(), it would recursively invoke the PrintFields method declared in Two, not the one declared in One.

Only by including an override modifier can a method override another method. In all other cases, a method with the same signature as an inherited method simply hides the inherited method. Consider the following code.

```
class One
{
  public virtual void FunctionOne()
  {}
}
class Two: One
{
  public virtual void FunctionOne()
  {}     // Warning, hiding inherited FunctionOne()
}
```

the FunctionOne method in Two does not include an override modifier and therefore does not override the FunctionOne method in One. Rather, the FunctionOne method in Two hides the method in One, and a warning is reported because the declaration does not include a new modifier. Consider the following code.

```
class One
{
  public virtual void FunctionOne()
  {}
}
class Two: One
{
  new private void FunctionOne()
  {}     // Hides One.FunctionOne within Two
}
class Three: Two
{
  public override void FunctionOne()
  {}   // Ok, overrides One.FunctionOne
}
```

the FunctionOne method in Two hides the virtual FunctionOne method inherited from One. Since the new FunctionOne in Two has private access, its scope only includes the class body of Two and does not extend to Three. The declaration of FunctionOne in Three is therefore permitted to override the FunctionOne inherited from One.

Sealed methods

When an instance method declaration includes a sealed modifier, the method is said to be a sealed method. A sealed method overrides an inherited virtual method with the same signature. Whereas a virtual method declaration introduces a new method, an override method declaration specializes an existing inherited virtual method by providing a new implementation of the method.

An override method can also be marked with the sealed modifier. Use of this modifier prevents a derived class from further overriding the method. The sealed modifier can only be used in combination with the override modifier. Consider the following code.

```
class One
{
  public virtual void FunctionOne()
  {
    Console.WriteLine("One.FunctionOne");
  }
  public virtual void FunctionTwo()
  {
    Console.WriteLine("One.FunctionTwo");
  }
```

```
}
class Two: One
{
  sealed override public void FunctionOne()
  {
    Console.WriteLine("Two.FunctionOne");
  }
  override public void FunctionTwo()
  {
    Console.WriteLine("Two.FunctionTwo");
  }
}
class Three: Two
{
  override public void FunctionTwo()
  {
    Console.WriteLine("Three.FunctionTwo");
  }
}
```

Here, the class Two provides two override methods: an FunctionOne method that has the sealed modifier and a FunctionTwo method that does not. Two's use of the sealed modifier prevents Three from further overriding FunctionOne.

Abstract methods

When an instance method declaration includes an abstract modifier, the method is said to be an abstract method. An abstract method is implicitly also a virtual method.

An abstract method declaration introduces a new virtual method but does not provide an implementation of the method. Instead, non-abstract derived classes are required to provide their own implementation by overriding the method. Because an abstract method provides no actual implementation, the method-body of an abstract method simply consists of a semicolon.

Abstract method declarations are only permitted in abstract classes. It is an error for an abstract method declaration to include either the static or virtual modifiers. Consider the following code.

```
public abstract class Shape
{
  public abstract void Paint(Graphics g, Rectangle r);
}
public class Ellipse: Shape
{
  public override void Paint(Graphics g, Rectangle r)
  {
    g.drawEllipse(r);
  }
}
public class Box: Shape
{
  public override void Paint(Graphics g, Rectangle r)
  {
    g.drawRect(r);
  }
}
```

Here, the Shape class defines the abstract notion of a geometrical shape object that can paint itself. The Paint method is abstract because there is no meaningful default implementation. The Ellipse and Box classes are concrete Shape implementations. Because theses classes are non-abstract, they are required to override the Paint method and provide an actual implementation.

It is an error for a base-access to reference an abstract method. Consider the following code.

```
class One
{
  public abstract void FunctionOne();
}
class Two: One
{
  public override void FunctionOne()
  {
    base.FunctionOne();
    // Error, base.FunctionOne is abstract
  }
}
```

Here, an error is reported for the base.FunctionOne() invocation because it references an abstract method.

An abstract method declaration is permitted to override a virtual method. This allows an abstract class to force re-implementation of the method in derived classes, and makes the original implementation of the method unavailable. Consider the following code.

```
class One
{
  public virtual void FunctionOne()
  {
    Console.WriteLine("One.FunctionOne");
  }
}
abstract class Two: One
{
  public abstract override void FunctionOne();
}
class Three: Two
{
  public override void FunctionOne()
  {
    Console.WriteLine("THree.FunctionOne");
  }
}
```

Here, the class One declares a virtual method, the class Two override this method with an abstract method, and the class Three overrides to provide its own implementation.

External methods

When a method declaration includes an extern modifier, the method is said to be an external method. External methods are implemented externally, using a language other than C#. Because an external method declaration provides no actual implementation, the method-body of an external method simply consists of a semicolon.

The extern modifier is typically used in conjunction with a DllImport attribute, allowing external methods to be implemented by DLLs (Dynamic Link Libraries). The execution environment may support other mechanisms whereby implementations of external methods can be provided.

It is an error for an external method declaration to also include the abstract modifier. When an external method includes a DllImport attribute, the method declaration must also include a static modifier.

The example listed below demonstrates use of the extern modifier and the DllImport attribute.

```
class Path
{
   [DllImport("kernel32", setLastError=true)]
   static extern bool CreateDirectory(string name,SecurityAttributes sa);
   [DllImport("kernel32", setLastError=true)]
   static extern bool RemoveDirectory(string name);
   [DllImport("kernel32", setLastError=true)]
   static extern int GetCurrentDirectory(int bufSize, StringBuilderbuf);
   [DllImport("kernel32", setLastError=true)]
   static extern bool SetCurrentDirectory(string name);
}
```

Method body

The method-body of a method declaration consists either of a block or a semicolon. Abstract and external method declarations do not provide a method implementation, and the method body of an abstract or external method simply consists of a semicolon. For all other methods, the method body is a block that contains the statements to execute when the method is invoked.

When the return type of a method is void, return statements in the method body are not permitted to specify an expression. If execution of the method body of a void method completes normally (that is, if control flows off the end of the method body), the method simply returns to the caller.

When the return type of a method is not void, each return statement in the method body must specify an expression of a type that is implicitly convertible to the return type. Execution of the method body of a value-returning method is required to terminate in a return statement that specifies an expression, or in a throw statement that throws an exception. It is an error if execution of the method body can complete normally. In other words, in a value-returning method, control is not permitted to flow off the end of the method body.

Consider the following code.

```
class One
{
   public int FunctionOne()
   {}     // Error, return value required
   public int FunctionTwo()
   {
      return 1;
   }
   public int FunctionThree(bool b)
   {
      if (b)
      {
         return 1;
      }
      else
      {
         return 0;
      }
   }
}
```

Here, the value-returning FunctionOne method is in error because control can flow off the end of the method body. The FunctionTwo and FunctionThree methods are correct because all possible execution paths end in a return statement that specifies a return value.

Student Activity 2

1. What do you understand by method? Give syntax to declare a method.
2. Differentiate between value and reference parameters.
3. What is the difference between static and instance method? Explain with the help of suitable example.
4. What do you mean by overriding a method? How we override a method in C#?
5. Differentiate Sealed Method and Abstract Method.

Properties

A property is a member that provides access to an attribute of an object or a class. Examples of properties include the length of a string, the size of a font, the caption of a window, the name of a customer, and so on. Properties are a natural extension of fields-both are named members with associated types, and the syntax for accessing fields and properties is the same. However, unlike fields, properties do not denote storage locations. Instead, properties have accessors that specify the statements to execute in order to read or write their values. Properties thus provide a mechanism for associating actions with the reading and writing of an object's attributes, and they furthermore permit such attributes to be computed.

Properties are declared using property-declarations. A property-declaration may include a set of attributes, a new modifier, a valid combination of the four access modifiers, and a valid combination of the static, virtual, override, and abstract modifiers. In addition, a property that includes the override modifier may also include the sealed modifier.

The static, virtual, override, and abstract modifiers are mutually exclusive except in one case. The abstract and override modifiers may be used together so that an abstract property can override a virtual one.

The type of a property declaration specifies the type of the property introduced by the declaration, and the member-name specifies the name of the property. Unless the property is an explicit interface member implementation, the member-name is simply an identifier. For an explicit interface member implementation, the member-name consists of an interface-type followed by a "." and an identifier.

The type of a property must be at least as accessible as the property itself.

The accessor-declarations, which must be enclosed in "{" and "}" tokens, declare the accessors of the property. The accessors specify the executable statements associated with reading and writing the property.

Even though the syntax for accessing a property is the same as that for a field, a property is not classified as a variable. Thus, it is not possible to pass a property as a ref or out parameter.

Static properties

When a property declaration includes a static modifier, the property is said to be a static property. When no static modifier is present, the property is said to be an instance property.

A static property is not associated with a specific instance, and it is an error to refer to this in the accessors of a static property. It is furthermore an error to include a virtual, abstract, or override modifier on a static property.

An instance property is associated with a given instance of a class, and this instance can be accessed as this in the accessors of the property. When a property is referenced in a member-access of the form E.M, if M is a static property, E must denote a type, and if M is an instance property, E must denote an instance.

Accessors

The accessor-declarations of a property specify the executable statements associated with reading and writing the property. The accessor declarations consist of a get-accessor-declaration, a set-accessor-declaration, or both. Each accessor declaration consists of the token get or set followed by an accessor-body. For abstract properties, the accessor-body for each accessor specified is simply a semicolon. For all other accessors, the accessor-body is a block which specifies the statements to execute when the accessor is invoked.

A get accessor corresponds to a parameterless method with a return value of the property type. Except as the target of an assignment, when a property is referenced in an expression, the get accessor of the property is invoked to compute the value of the property. In particular, all return statements in the body of a get accessor must specify an expression that is implicitly convertible to the property type. Furthermore, a get accessor is required to terminate in a return statement or a throw statement, and control is not permitted to flow off the end of the get accessor's body.

A set accessor corresponds to a method with a single value parameter of the property type and a void return type. The implicit parameter of a set accessor is always named value. When a property is referenced as the target of an assignment, the set accessor is invoked with an argument that provides the new value. The body of a set accessor must conform to the rules for void methods. In particular, return statements in the set accessor body are not permitted to specify an expression. Since a set accessor implicitly has a parameter named value, it is an error for a local variable declaration in a set accessor to use that name.

Based on the presence or absence of the get and set accessors, a property is classified as follows:

- A property that includes both a get accessor and a set accessor is said to be a read-write property.
- A property that has only a get accessor is said to be a read-only property. It is an error for a read-only property to be the target of an assignment.
- A property that has only a set accessor is said to be a write-only property. Except as the target of an assignment, it is an error to reference a write-only property in an expression.

Consider the following code.

```
public class Button: Control
{
  private string caption;
  public string Caption
  {
    get
    {
      return caption;
    }
    set {
      if (caption != value)
      {
        caption = value;
        Repaint();
      }
    }
  }
  public override void Paint(Graphics g, Rectangle r)
  {
    // Painting code goes here
  }
}
```

Here, the Button control declares a public Caption property. The get accessor of the Caption property returns the string stored in the private caption field. The set accessor checks if the new

value is different from the current value, and if so, it stores the new value and repaints the control. Properties often follow the pattern shown above: The get accessor simply returns a value stored in a private field, and the set accessor modifies the private field and then performs any additional actions required to fully update the state of the object.

Given the Button class above, the following is an example of use of the Caption property:

```
Button okButton = new Button();
okButton.Caption = "OK";                  // Invokes set accessor
string s = okButton.Caption;              // Invokes get accessor
```

Here, the set accessor is invoked by assigning a value to the property, and the get accessor is invoked by referencing the property in an expression.

When a derived class declares a property by the same name as an inherited property, the derived property hides the inherited property with respect to both reading and writing. Consider the following code.

```
class One
{
  public int P;
{
     set {…}
  }
}
class B: A
{
  new public int P;
{
     get {…}
  }
}
```

Here, the P property in B hides the P property in A with respect to both reading and writing. Thus, in the statements

```
B b = new B();
b.P = 1;          // Error, B.P is read-only
((A)b).P = 1;     // Ok, reference to A.P
```

the assignment to b.P causes an error to be reported, since the read-only P property in B hides the write-only P property in A. Note, however, that a cast can be used to access the hidden P property.

Unlike public fields, properties provide a separation between an object's internal state and its public interface. Consider the following code.

```
class Label
{
  private int x, y;
  private string caption;
  public Label(int x, int y, string caption)
  {
    this.x = x;
    this.y = y;
    this.caption = caption;
  }
  public int X
  {
  get { return x; }
```

```
    }
    public int Y
    {
    get { return y; }
    }
    public Point Location
    {
    get { return new Point(x, y); }
    }
    public string Caption
    {
    get { return caption; }
    }
}
```

Here, the Label class uses two int fields, x and y, to store its location. The location is publicly exposed both as an X and a Y property and as a Location property of type Point. If, in a future version of Label, it becomes more convenient to store the location as a Point internally, the change can be made without affecting the public interface of the class:

```
class Label
{
  private Point location;
  private string caption;
  public Label(int x, int y, string caption)
  {
    this.location = new Point(x, y);
    this.caption = caption;
  }
  public int X
  {
    get { return location.x; }
  }
  public int Y
  {
    get { return location.y; }
  }
  public Point Location
  {
    get { return location; }
  }
  public string Caption
  {
    get { return caption; }
  }
}
```

Had x and y instead been public readonly fields, it would have been impossible to make such a change to the Label class. Exposing state through properties is not necessarily any less efficient than exposing fields directly. In particular, when a property is non-virtual and contains only a small amount of code, the execution environment may replace calls to accessors with the actual code of the accessors. This process is known as inlining, and it makes property access as efficient as field access, yet preserves the increased flexibility of properties.

Since invoking a get accessor is conceptually equivalent to reading the value of a field, it is considered bad programming style for get accessors to have observable side-effects. In the code

example listed below

```
class Counter
{
  private int next;
  public int Next
  {
  get { return next++;
}}
}
```

the value of the Next property depends on the number of times the property has previously been accessed. Thus, accessing the property produces an observable side-effect, and the property should instead be implemented as a method.

The "no side-effects" convention for get accessors doesn't mean that get accessors should always be written to simply return values stored in fields. Indeed, get accessors often compute the value of a property by accessing multiple fields or invoking methods. However, a properly designed get accessor performs no actions that cause observable changes in the state of the object.

Properties can be used to delay initialization of a resource until the moment it is first referenced (see the example code listed below).

```
using System.IO;
public class Console
{
  private static TextReader reader;
  private static TextWriter writer;
  private static TextWriter error;
  public static TextReader In
  {
    get
    {
    if (reader == null)
    {
    reader = new StreamReader(File.OpenStandardInput());
    }
    return reader;
  }
  }
  public static TextWriter Out
  {
    get
{
    if (writer == null)
  {
  writer = new StreamWriter(File.OpenStandardOutput());
  }
  return writer;
}
}
  public static TextWriter Error
  {
    get {
      if (error == null)
```

```
    {
    error = new StreamWriter(File.OpenStandardError());
    }
    return error;
      }
  }
  }
```

The Console class contains three properties, In, Out, and Error, that represent the standard input, output, and error devices, respectively. By exposing these members as properties, the Console class can delay their initialization until they are actually used. For example, upon first referencing the Out property, as in

```
Console.Out.WriteLine("hello, there");
```

the underlying TextWriter for the output device is created. But if the application makes no reference to the In and Error properties, then no objects are created for those devices.

Virtual, sealed, override, and abstract accessors

A property declaration may include a valid combination of the static, virtual, override, and abstract modifiers. A property that includes the override modifier may also include the sealed modifier.

The static, virtual, override, and abstract modifiers are mutually exclusive except in one case. The abstract and override modifiers may be used together so that an abstract property can override a virtual one. A virtual property declaration specifies that the accessors of the property are virtual. The virtual modifier applies to both accessors of a read-write property-it is not possible for only one accessor of a read-write property to be virtual.

An abstract property declaration specifies that the accessors of the property are virtual, but does not provide an actual implementation of the accessors. Instead, non-abstract derived classes are required to provide their own implementation for the accessors by overriding the property. Because an accessor for an abstract property declaration provides no actual implementation, its accessor-body simply consists of a semicolon.

A property declaration that includes both the abstract and override modifiers specifies that the property is abstract and overrides a base property. The accessors of such a property are also abstract.

Abstract property declarations are only permitted in abstract classes.The accessors of an inherited virtual property can be overridden in a derived class by including a property declaration that specifies an override directive. This is known as an overriding property declaration. An overriding property declaration does not declare a new property. Instead, it simply specializes the implementations of the accessors of an existing virtual property.

An overriding property declaration must specify the exact same accessibility modifiers, type, and name as the inherited property. If the inherited property has only a single accessor (i.e. if the inherited property is read-only or write-only), the overriding property can and must include only that accessor. If the inherited property includes both accessors (i.e. if the inherited property is read-write), the overriding property can include either a single accessor or both accessors.

An overriding property declaration may include the sealed modifier. Use of this modifier prevents a derived class from further overriding the property. The accessors of a sealed property are also sealed. It is an error for an overriding property declaration to include a new modifier.

Except for differences in declaration and invocation syntax, virtual, sealed, override, and abstract accessors behave exactly like a virtual, sealed, override and abstract methods.

A get accessor corresponds to a parameterless method with a return value of the property type and the same modifiers as the containing property.

A set accessor corresponds to a method with a single value parameter of the property type, a void return type, and the same modifiers as the containing property. Consider the following code.

```
abstract class One
{
  int y;
  public virtual int X
  {
    get { return 0; }
}
public virtual int Y
{
  get { return y; }
  set { y = value; }
}
public abstract int Z { get; set; }
}
```

Here, X is a virtual read-only property, Y is a virtual read-write property, and Z is an abstract read-write property. Because Z is abstract, the containing class A must also be declared abstract. A class that derives from A is show below:

```
class B: A
{
  int z;
  public override int X
  {
  get { return base.X + 1; }
  }
public override int Y
{
  set { base.Y = value < 0? 0: value; }
}
public override int Z
{
  get { return z; }
  set { z = value; }
}
}
```

Here, the declarations of X, Y, and Z are overriding property declarations. Each property declaration exactly matches the accessibility modifiers, type, and name of the corresponding inherited property. The get accessor of X and the set accessor of Y use the base keyword to access the inherited accessors. The declaration of Z overrides both abstract accessors-thus, there are no outstanding abstract function members in B, and B is permitted to be a non-abstract class.

4.5 EVENTS

An event is a member that enables an object or class to provide notifications. Clients can attach executable code for events by supplying event handlers.

Events are declared using event-declarations. An event-declaration may include a set of attributes, a new modifier, a valid combination of the four access modifiers, and a valid combination of the static, virtual, override, and abstract modifiers. In addition, an event that includes the override modifier may also include the sealed modifier.

The static, virtual, override, and abstract modifiers are mutually exclusive except in one case. The abstract and override modifiers may be used together so that an abstract event can override a virtual one.

An event declaration may include event-accessor-declarations, or may rely on the compiler to supply such accessors automatically. An event declaration that omits event-accessor-declarations defines one or more events-one for each of the variable-declarators. The attributes and modifiers apply to all of the members declared by such an event-declaration.

An abstract event is declared with an event-declaration that omits event-accessor-declarations. It is an error for an event-declaration to include both the abstract modifier and event-accessor-declarations. The type of an event declaration must be a delegate-type, and that delegate-type must be at least as accessible as the event itself.

An event can be used as the left hand operand of the += and -= operators. These operators are used to attach or remove event handlers to or from an event, and the access modifiers of the event control the contexts in which the operations are permitted.

Since += and -= are the only operations that are permitted on an event outside the type that declares the event, external code can add and remove handlers for an event, but cannot in any other way obtain or modify the underlying list of event handlers.

Within the program text of the class or struct that contains the declaration of an event, certain events can be used like fields. To be used in this way, an event must not be abstract, and must not explicitly include event-accessor-declarations. Such an event can be used in any context that permits a field.

Consider the following code.

```
public delegate void EventHandler(object sender, EventArgs e);
public class Button: Control
{
  public event EventHandler Click;
  protected void OnClick(EventArgs e)
  {
    if (Click != null) Click(this, e);
  }
  public void Reset()
  {
  Click = null;
  }
}
```

Click is used as a field within the Button class. As the example demonstrates, the field can be examined, modified, and used in delegate invocation expressions. The OnClick method in the Button class "raises" the Click event. The notion of raising an event is precisely equivalent to invoking the delegate represented by the event-thus, there are no special language constructs for raising events. Note that the delegate invocation is preceded by a check that ensures the delegate is non-null.

Outside the declaration of the Button class, the Click member can only be used on the left hand side of the += and -= operators, as in

```
b.Click += new EventHandler(...);
```

which appends a delegate to the invocation list of the Click event, and

```
b.Click -= new EventHandler(...);
```

which removes a delegate from the invocation list of the Click event.

In an operation of the form x += y or x -= y, when x is an event and the reference takes place outside the type that contains the declaration of x, the result of the operation is void (as opposed to the

value of x after the assignment). This rule prohibits external code from indirectly examining the underlying delegate of an event.

The following example shows how event handlers are attached to instances of the Button class above:

```
public class LoginDialog: Form
{
  Button OkButton;
  Button CancelButton;
  public LoginDialog()
  {
  OkButton = new Button(...);
  OkButton.Click += new EventHandler(OkButtonClick);
  CancelButton = new Button(...);
  CancelButton.Click += new EventHandler(CancelButtonClick);
  }
  void OkButtonClick(object sender, EventArgs e)
  {
    // Handle OkButton.Click event
  }
  void CancelButtonClick(object sender, EventArgs e)
  {
  // Handle CancelButton.Click event
  }
}
```

Here, the LoginDialog constructor creates two Button instances and attaches event handlers to the Click events.

Event accessors

Event declarations typically omit event-accessor-declarations, as in the Button example above. In cases where the storage cost of one field per event is not acceptable, a class can include event-accessor-declarations and use a private mechanism for storing the list of event handlers.

The event-accessor-declarations of an event specify the executable statements associated with adding and removing event handlers.

The accessor declarations consist of an add-accessor-declaration and a remove-accessor-declaration. Each accessor declaration consists of the token add or remove followed by a block. The block associated with an add-accessor-declaration specifies the statements to execute when an event handler is added, and the block associated with a remove-accessor-declaration specifies the statements to execute when an event handler is added.

An event accessor, whether an add-accessor-declaration or a remove-accessor-declaration, corresponds to a method with a single value parameter of the event type and a void return type. The implicit parameter of an event accessor is always named value. When an event is used in an event assignment, the appropriate event accessor is used. If the assignment operator is += then the add accessor is used, and if the assignment operator is -= then the remove accessor is used. In either case, the right hand side of the assignment operator is used as the argument to the event accessor. The block of an add-accessor-declaration or a remove-accessor-declaration must conform to the rules for void methods. In particular, return statements in such a block are not permitted to specify an expression.

Since an event accessor implicitly has a parameter named value, it is an error for a local variable declaration in an event accessor to use that name. Consider the following code.

```
class Control: Component
{
   // Unique keys for events
   static readonly object mouseDownEventKey = new object();
   static readonly object mouseUpEventKey = new object();
   // Return event handler associated with key
   protected Delegate GetEventHandler(object key) {...}
   // Add event handler associated with key
   protected void AddEventHandler(object key, Delegate handler) {...}
   // Remove event handler associated with key
   protected void RemoveEventHandler(object key, Delegate handler){...}
   // MouseDown event
   public event MouseEventHandler MouseDown
   {
      add { AddEventHandler(mouseDownEventKey, value); }
      remove { AddEventHandler(mouseDownEventKey, value); }
   }
   // MouseUp event
   public event MouseEventHandler MouseUp
   {
      add { AddEventHandler(mouseUpEventKey, value); }
      remove { AddEventHandler(mouseUpEventKey, value); }
   }
}
```

Here, the Control class implements an internal storage mechanism for events. The AddEventHandler method associates a delegate value with a key, the GetEventHandler method returns the delegate currently associated with a key, and the RemoveEventHandler method removes a delegate as an event handler for the specified event. Presumably the underlying storage mechanism is designed such that there is no cost for associating a null delegate value with a key, and thus unhandled events consume no storage.

Static events

When an event declaration includes a static modifier, the event is said to be a static event. When no static modifier is present, the event is said to be an instance event.

A static event is not associated with a specific instance, and it is an error to refer to this in the accessors of a static event. It is furthermore an error to include a virtual, abstract, or override modifier on a static event.

An instance event is associated with a given instance of a class, and this instance can be accessed as this in the accessors of the event. When an event is referenced in a member-access of the form E.M, if M is a static event, E must denote a type, and if M is an instance event, E must denote an instance.

Virtual, sealed, override, and abstract accessors

An event declaration may include a valid combination of the static, sealed, virtual, override, and abstract modifiers. An event that includes the override modifier may also include the sealed modifier.

The static, virtual, override, and abstract modifiers are mutually exclusive except in one case. The abstract and override modifiers may be used together so that an abstract event can override a virtual one.

A virtual event declaration specifies that the accessors of the event are virtual. The virtual modifier applies to both accessors of an event. An abstract event declaration specifies that the accessors of the event are virtual, but does not provide an actual implementation of the accessors. Instead, non-abstract derived classes are required to provide their own implementation for the accessors by overriding the event. Because an accessor for an abstract event declaration provides no actual implementation, its accessor-body simply consists of a semicolon.

An event declaration that includes both the abstract and override modifiers specifies that the event is abstract and overrides a base property. The accessors of such an event are also abstract.

Abstract event declarations are only permitted in abstract classes.

The accessors of an inherited virtual event can be overridden in a derived class by including an event declaration that specifies an override directive. This is known as an overriding event declaration. An overriding event declaration does not declare a new event. Instead, it simply specializes the implementations of the accessors of an existing virtual event.

An overriding event declaration must specify the exact same accessibility modifiers, type, and name as the inherited event.

An overriding event declaration may include the sealed modifier. Use of this modifier prevents a derived class from further overriding the event. The accessors of a sealed event are also sealed. It is an error for an overriding event declaration to include a new modifier.

Except for differences in declaration and invocation syntax, virtual, override, and abstract accessors behave exactly like a virtual, sealed, override and abstract methods. Each accessor corresponds to a method with a single value parameter of the event type, a void return type, and the same modifiers as the containing event.

4.6 INDEXERS

An indexer is a member that enables an object to be indexed in the same way as an array. Indexers are declared using indexer-declarations. An indexer-declaration may include a set of attributes, a new modifier, a valid combination of the four access modifiers, and a valid combination of the virtual, override, and abstract modifiers. In addition, an indexer that includes the override modifier may also include the sealed modifier.

The static, virtual, override, and abstract modifiers are mutually exclusive except in one case. The abstract and override modifiers may be used together so that an abstract indexer can override a virtual one.

The type of an indexer declaration specifies the element type of the indexer introduced by the declaration. Unless the indexer is an explicit interface member implementation, the type is followed by the keyword this. For an explicit interface member implementation, the type is followed by an interface-type, a ".", and the keyword this. Unlike other members, indexers do not have user-defined names.

The formal-parameter-list specifies the parameters of the indexer. The formal parameter list of an indexer corresponds to that of a method, except that at least one parameter must be specified, and that the ref and out parameter modifiers are not permitted. The type of an indexer and each of the types referenced in the formal-parameter-list must be at least as accessible as the indexer itself.

The accessor-declarations, which must be enclosed in "{" and "}" tokens, declare the accessors of the indexer. The accessors specify the executable statements associated with reading and writing indexer elements.

Even though the syntax for accessing an indexer element is the same as that for an array element, an indexer element is not classified as a variable. Thus, it is not possible to pass an indexer element as a ref or out parameter.

The formal parameter list of an indexer defines the signature of the indexer. Specifically, the signature of an indexer consists of the number and types of its formal parameters. The element type is not part of an indexer's signature, nor are the names of the formal parameters.

The signature of an indexer must differ from the signatures of all other indexers declared in the same class. Indexers and properties are very similar in concept, but differ in the following ways:

- A property is identified by its name, whereas an indexer is identified by its signature.
- A property is accessed through a simple-name or a member-access, whereas an indexer element is accessed through an element-access.
- A property can be a static member, whereas an indexer is always an instance member.
- A get accessor of a property corresponds to a method with no parameters, whereas a get accessor of an indexer corresponds to a method with the same formal parameter list as the indexer.
- A set accessor of a property corresponds to a method with a single parameter named value, whereas a set accessor of an indexer corresponds to a method with the same formal parameter list as the indexer, plus an additional parameter named value.
- It is an error for an indexer accessor to declare a local variable with the same name as an indexer parameter.
- In an overriding property declaration, the inherited property is accessed using the syntax base.P, where P is the property name. In an overriding indexer declaration, the inherited indexer is accessed using the syntax base[E], where E is a comma separated list of expressions.

The example below declares a BitArray class that implements an indexer for accessing the individual bits in the bit array.

```
class BitArray
{
  int[] bits;
  int length;
  public BitArray(int length)
  {
    if (length < 0) throw new ArgumentException();
    bits = new int[((length - 1) >> 5) + 1];
    this.length = length;
  }
  public int Length
  {
    get { return length; }
  }
  public bool this[int index]
  {
    get {
    if (index < 0 || index >= length)
    {
      throw new IndexOutOfRangeException();
    }
    return (bits[index >> 5] & 1 << index) != 0;
  }
  set {
    if (index < 0 || index >= length)
    {
      throw new IndexOutOfRangeException();
    }
```

```
        if (value)
        {
          bits[index >> 5] |= 1 << index;
        }
        else {
          bits[index >> 5] &= ~(1 << index);
        }
      }
      }
  }
```

An instance of the BitArray class consumes substantially less memory than a corresponding bool[ ] (each value occupies only one bit instead of one byte), but it permits the same operations as a bool[ ].

The following CountPrimes class uses a BitArray and the classical "sieve" algorithm to compute the number of primes between 1 and a given maximum:

```
class CountPrimes
{
  static int Count(int max)
  {
    BitArray flags = new BitArray(max + 1);
    int count = 1;
    for (int i = 2; i <= max; i++)
{
    if (!flags[i])
{
    for (int j = i * 2; j <= max; j += i) flags[j] = true;
    count++;
  }
}
  return count;
}
  static void Main(string[] args)
  {
  int max = int.Parse(args[0]);
  int count = Count(max);
  Console.WriteLine("Found {0} primes between 1 and {1}", count, max);
  }
}
```

Note that the syntax for accessing elements of the BitArray is precisely the same as for a bool[].

Operators

An operator is a member that defines the meaning of an expression operator that can be applied to instances of the class. Operators are declared using operator-declarations. There are three categories of operators: Unary operators, binary operators, and conversion operators.

The following rules apply to all operator declarations:

- An operator declaration must include both a public and a static modifier, and is not permitted to include any other modifiers.
- The parameter(s) of an operator must be value parameters. It is an error to for an operator declaration to specify ref or out parameters.
- The signature of an operator must differ from the signatures of all other operators declared in the same class.

All types referenced in an operator declaration must be at least as accessible as the operator itself.

Each operator category imposes additional restrictions, as described in the following sections.

Like other members, operators declared in a base class are inherited by derived classes. Because operator declarations always require the class or struct in which the operator is declared to participate in the signature of the operator, it is not possible for an operator declared in a derived class to hide an operator declared in a base class. Thus, the new modifier is never required, and therefore never permitted, in an operator declaration.

For all operators, the operator declaration includes a block which specifies the statements to execute when the operator is invoked. The block of an operator must conform to the rules for value-returning methods.

Unary operators

The following rules apply to unary operator declarations, where T denotes the class or struct type that contains the operator declaration:

- A unary +, -, !, or ~ operator must take a single parameter of type T and can return any type.
- A unary ++ or -- operator must take a single parameter of type T and must return type T.
- A unary true or false operator must take a single parameter of type T and must return type bool.

The signature of a unary operator consists of the operator token (+, -, !, ~, ++, --, true, or false) and the type of the single formal parameter. The return type is not part of a unary operator's signature, nor is the name of the formal parameter.

The true and false unary operators require pair-wise declaration. An error occurs if a class declares one of these operators without also declaring the other.

Binary operators

A binary operator must take two parameters, at least one of which must be of the class or struct type in which the operator is declared. A binary operator can return any type.

The signature of a binary operator consists of the operator token (+, -, *, /, %, &, |, ^, <<, >>, ==, !=, >, <, >=, or <=) and the types of the two formal parameters. The return type is not part of a binary operator's signature, nor are the names of the formal parameters.

Certain binary operators require pair-wise declaration. For every declaration of either operator of a pair, there must be a matching declaration of the other operator of the pair. Two operator declarations match when they have the same return type and the same type for each parameter. The following operators require pair-wise declaration:

```
operator == and operator !=
operator > and operator <
operator >= and operator <=
```

Conversion operators

A conversion operator declaration introduces a user-defined conversion which augments the pre-defined implicit and explicit conversions.

A conversion operator declaration that includes the implicit keyword introduces a user-defined implicit conversion. Implicit conversions can occur in a variety of situations, including function member invocations, cast expressions, and assignments.

A conversion operator declaration that includes the explicit keyword introduces a user-defined explicit conversion. Explicit conversions can occur in cast expressions.

A conversion operator converts from a source type, indicated by the parameter type of the conversion operator, to a target type, indicated by the return type of the conversion operator. A

class or struct is permitted to declare a conversion from a source type S to a target type T provided all of the following are true:

- S and T are different types.
- Either S or T is the class or struct type in which the operator declaration takes place.
- Neither S nor T is object or an interface-type.
- T is not a base class of S, and S is not a base class of T.

From the second rule it follows that a conversion operator must either convert to or from the class or struct type in which the operator is declared. For example, it is possible for a class or struct type C to define a conversion from C to int and from int to C, but not from int to bool.

It is not possible to redefine a pre-defined conversion. Thus, conversion operators are not allowed to convert from or to object because implicit and explicit conversions already exist between object and all other types. Likewise, neither of the source and target types of a conversion can be a base type of the other, since a conversion would then already exist.

User-defined conversions are not allowed to convert from or to interface-types. This restriction in particular ensures that no user-defined transformations occur when converting to an interface-type, and that a conversion to an interface-type succeeds only if the object being converted actually implements the specified interface-type.

The signature of a conversion operator consists of the source type and the target type. (Note that this is the only form of member for which the return type participates in the signature.) The implicit or explicit classification of a conversion operator is not part of the operator's signature. Thus, a class or struct cannot declare both an implicit and an explicit conversion operator with the same source and target types.

In general, user-defined implicit conversions should be designed to never throw exceptions and never lose information. If a user-defined conversion can give rise to exceptions (for example because the source argument is out of range) or loss of information (such as discarding high-order bits), then that conversion should be defined as an explicit conversion.

Consider the following code.

```
public struct Digit
{
  byte value;
public Digit(byte value)
{
  if (value < 0 || value > 9) throw new ArgumentException();
  this.value = value;
}
public static implicit operator byte(Digit d)
{
  return d.value;
}
public static explicit operator Digit(byte b)
{
  return new Digit(b);
}
}
```

Here, the conversion from Digit to byte is implicit because it never throws exceptions or loses information, but the conversion from byte to Digit is explicit since Digit can only represent a subset of the possible values of a byte.

Student Activity 2

1. What are properties? How to declare and access property in C#.
2. Can properties be static? If Yes, explain with example.
3. What are events? Explain static and instance events.
4. What are indexes? How it is different from properties?
5. Explain the various types of operators.

4.7 CONSTRUCTORS

An instance constructor is a member that implements the actions required to initialize an instance of a class. Constructors are declared using constructor-declarations. A constructor-declaration may include a set of attributes and a valid combination of the four access modifiers.

The identifier of a constructor-declarator must name the class in which the constructor is declared. If any other name is specified, an error occurs. The optional formal-parameter-list of a constructor is subject to the same rules as the formal-parameter-list of a method. The formal parameter list defines the signature of a constructor and governs the process whereby overload resolution selects a particular constructor in an invocation. Each of the types referenced in the formal-parameter-list of a constructor must be at least as accessible as the constructor itself.

The optional constructor-initializer specifies another constructor to invoke before executing the statements given in the block of this constructor.

The block of a constructor declaration specifies the statements to execute in order to initialize a new instance of the class. This corresponds exactly to the block of an instance method with a void return type. Constructors are not inherited. Thus, a class has no other constructors than those that are actually declared in the class. If a class contains no constructor declarations, a default constructor is automatically provided. Constructors are invoked by object-creation-expressions and through constructor-initializers.

Constructor initializers

All constructors (except for the constructors of class object) implicitly include an invocation of another constructor immediately before the first statement in the block of the constructor. The constructor to implicitly invoke is determined by the constructor-initializer:

- A constructor initializer of the form base(…) causes a constructor from the direct base class to be invoked. The set of candidate constructors consists of all accessible constructors declared in the direct base class. If the set of candidate constructors is empty, or if a single best constructor cannot be identified, an error occurs.
- A constructor initializer of the form this(…) causes a constructor from the class itself to be invoked. The set of candidate constructors consists of all accessible constructors declared in the class itself. If the set of candidate constructors is empty, or if a single best constructor cannot be identified, an error occurs. If a constructor declaration includes a constructor initializer that invokes the constructor itself, an error occurs.

If a constructor has no constructor initializer, a constructor initializer of the form base() is implicitly provided. Thus, a constructor declaration of the form

```
C(…) {…}
```

is exactly equivalent to

```
C(…): base() {…}
```

The scope of the parameters given by the formal-parameter-list of a constructor declaration includes the constructor initializer of that declaration. Thus, a constructor initializer is permitted to access the parameters of the constructor. Consider the following code.

```
class One
{
  public One(int x, int y)
{}
}
class Two: One
{
  public Two(int x, int y): base(x + y, x - y)
{}
}
```

A constructor initializer cannot access the instance being created. It is therefore an error to reference this in an argument expression of the constructor initializer, as it is an error for an argument expression to reference any instance member through a simple-name.

Constructor execution

It is useful to think of instance variable initializers and constructor initializers as statements that are automatically inserted before the first statement in the block of a constructor. Consider the following code.

```
using System.Collections;
class One
{
  int x = 1, y = -1, count;
  public One()
{
    count = 0;
  }
public One(int n)
{
    count = n;
  }
}
class Two: One
{
  double sqrt2 = Math.Sqrt(2.0);
  ArrayList items = new ArrayList(100);
  int max;
  public Two(): this(100)
{
    items.Add("default");
  }
  public Two(int n): base(n - 1)
{
    max = n;
  }
}
```

This code contains several variable initializers and also contains constructor initializers of both forms (base and this). The example corresponds to the code shown below, where each comment indicates an automatically inserted statement (the syntax used for the automatically inserted constructor invocations isn't valid, but merely serves to illustrate the mechanism).

```
using System.Collections;
class One
{
  int x, y, count;
  public One()
  {
    x = 1;   // Variable initializer
    y = -1;     // Variable initializer
    object();   // Invoke object() constructor
    count = 0;
  }
  public A(int n)
  {
    x = 1;      // Variable initializer
    y = -1;       // Variable initializer
    object();     // Invoke object() constructor
    count = n;
  }
}
class Two: One
{
  double sqrt2;
  ArrayList items;
  int max;
  public Two(): this(100)
  {
    Two(100);     // Invoke Two(int) constructor
    items.Add("default");
  }
  public Two(int n): base(n - 1)
  {
    sqrt2 = Math.Sqrt(2.0);     // Variable initializer
    items = new ArrayList(100); // Variable initializer
    One(n - 1);        // Invoke One(int) constructor
    max = n;
  }
}
```

Note that variable initializers are transformed into assignment statements, and that these assignment statements are executed before the invocation of the base class constructor. This ordering ensures that all instance fields are initialized by their variable initializers before any statements that have access to the instance are executed. Consider the following code.

```
class One
{
  public One()
{
    PrintFields();
  }
public virtual void PrintFields()
{}
}
class Two: One
{
```

```
    int x = 1;
    int y;
  public Two()
  {
    y = -1;
  }
  public override void PrintFields()
  {
      Console.WriteLine("x = {0}, y = {1}", x, y);
    }
  }
```

When new Two() is used to create an instance of Two, the following output is produced:

```
x = 1, y = 0
```

The value of x is 1 because the variable initializer is executed before the base class constructor is invoked. However, the value of y is 0 (the default value of an int) because the assignment to y is not executed until after the base class constructor returns.

Default constructors

If a class contains no constructor declarations, a default constructor is automatically provided. The default constructor simply invokes the parameterless constructor of the direct base class. If the direct base class does not have an accessible parameterless constructor, an error occurs. If the class is abstract then the declared accessibility for the default constructor is protected. Otherwise, the declared accessibility for the default constructor is public. Thus, the default constructor is always of the form

```
protected C(): base() {}
or
public C(): base() {}
where C is the name of the class.
Consider the following code.

class Message
{
    object sender;
    string text;
}
```

Here, a default constructor is provided because the class contains no constructor declarations. Thus, the example is precisely equivalent to

```
  class Message
  {
    object sender;
    string text;
  public Message(): base()
  {}
  }
```

Private constructors

When a class declares only private constructors it is not possible for other classes to derive from the class or create instances of the class (an exception being classes nested within the class). Private constructors are commonly used in classes that contain only static members. Consider the following code.

```
  public class Trig
  {
```

```
  private Trig()
{}   // Prevent instantiation
public const double PI = 3.14159265358979323846;
public static double Sin(double x) {…}
public static double Cos(double x) {…}
public static double Tan(double x) {…}
}
```

The Trig class provides a grouping of related methods and constants, but is not intended to be instantiated. It therefore declares a single private constructor. At least one private constructor must be declared to suppress the automatic generation of a default constructor.

Optional constructor parameters

The this(…) form of constructor initializer is commonly used in conjunction with overloading to implement optional constructor parameters. Consider the following code.

```
class Text
{
  public Text(): this(0, 0, null)
{}
public Text(int x, int y): this(x, y, null)
{}
public Text(int x, int y, string s)
{
  // Actual constructor implementation
}
}
```

Here, the first two constructors merely provide the default values for the missing arguments. Both use a this(…) constructor initializer to invoke the third constructor, which actually does the work of initializing the new instance. The effect is that of optional constructor parameters:

```
Text t1 = new Text();         // Same as Text(0, 0, null)
Text t2 = new Text(5, 10);      // Same as Text(5, 10, null)
Text t3 = new Text(5, 20, "Hello");
```

Static constructors

A static constructor is a member that implements the actions required to initialize a class. Static constructors are declared using static-constructor-declarations. A static-constructor-declaration may include a set of attributes.

The identifier of a static-constructor-declaration must name the class in which the static constructor is declared. If any other name is specified, an error occurs. The block of a static constructor declaration specifies the statements to execute in order to initialize the class. This corresponds exactly to the block of a static method with a void return type. Static constructors are not inherited.

Class loading is the process by which a class is prepared for use in the runtime environment. The loading process is mostly implementation-dependent, though several guarantees are provided:

- A class is loaded before any instance of the class is created.
- A class is loaded before any of its static members are referenced.
- A class is loaded before any types that derive from it are loaded.
- A class cannot be loaded more than once during a single execution of a program.
- If a class has a static constructor then it is automatically called when the class is loaded. Static constructors cannot be invoked explicitly.

Consider the following example.

```
class TestClass
{
  static void Main()
{
    One.FunctionOne();
    Two.FunctionOne();
  }
}
class One
{
  static One()
{
    Console.WriteLine("Init One");
  }
public static void FunctionOne()
{
  Console.WriteLine("One.FunctionOne");
}
}
class Two
{
  static Two()
{
    Console.WriteLine("Init Two");
  }
public static void FunctionOne()
{
  Console.WriteLine("Two.FunctionOne");
}
}
```

It could produce either the output:

```
Init One
One.FunctionOne
Init Two
Two.FunctionOne
```

or the output:

```
Init Two
Init One
One.FunctionOne
Two.FunctionOne
```

because the exact ordering of loading and therefore of static constructor execution is not defined.

Consider the following code.

```
class TestClass
{
  static void Main()
{
    Console.WriteLine("1");
    Two.FunctionTwo();
    Console.WriteLine("2");
  }
```

```
}
class One
{
  static One()
{
    Console.WriteLine("Init One");
  }
}
class Two: One
{
  static Two()
{
    Console.WriteLine("Init Two");
  }
  public static void FunctionTwo()
{
    Console.WriteLine("Two.FunctionTwo");
  }
}
```

It is guaranteed to produce the output:

```
Init One
Init Two
Two.FunctionTwo
```

because the static constructor for the class One must execute before the static constructor of the class Two, which derives from it.

It is possible to construct circular dependencies that allow static fields with variable initializers to be observed in their default value state. Consider the following code.

```
class One
{
  public static int X = Two.Y + 1;
}
class Two
{
  public static int Y = One.X + 1;
static void Main()
{
  Console.WriteLine("X = {0}, Y = {1}", One.X, Two.Y);
}
}
```

It produces the output

X = 1, Y = 2

To execute the Main method, the system first loads class Two. The static constructor of Two proceeds to compute the initial value of Y, which recursively causes One to be loaded because the value of One.X is referenced. The static constructor of One in turn proceeds to compute the initial value of X, and in doing so fetches the default value of Y, which is zero. One.X is thus initialized to 1. The process of loading One then completes, returning to the calculation of the initial value of Y, the result of which becomes 2.

Had the Main method instead been located in class One, the example would have produced the output

X = 2, Y = 1

Circular references in static field initializers should be avoided since it is generally not possible to determine the order in which classes containing such references are loaded.

4.8 DESTRUCTORS

A destructor is a member that implements the actions required to destruct an instance of a class. Destructors are declared using destructor-declarations. A destructor-declaration may include a set of attributes.

The identifier of a destructor-declarator must name the class in which the destructor is declared. If any other name is specified, an error occurs.

The block of a destructor declaration specifies the statements to execute in order to destruct an instance of the class. This corresponds exactly to the block of an instance method with a void return type. Destructors are not inherited. Thus, a class has no other destructors than those that are actually declared in the class.

Destructors are invoked automatically, and cannot be invoked explicitly. An instance becomes eligible for destruction when it is no longer possible for any code to use the instance. Execution of the destructor for the instance may occur at any time after the instance becomes eligible for destruction. When an instance is destructed, the destructors in an inheritance chain are called in order, from most derived to least derived.

Expressions and Access Modifiers

This access

A this-access consists of the reserved word this. A this-access is permitted only in the block of a constructor, an instance method, or an instance accessor. It has one of the following meanings:

- When this is used in a primary-expression within a constructor of a class, it is classified as a value. The type of the value is the class within which the reference occurs, and the value is a reference to the object being constructed.
- When this is used in a primary-expression within an instance method or instance accessor of a class, it is classified as a value. The type of the value is the class within which the reference occurs, and the value is a reference to the object for which the method or accessor was invoked.
- When this is used in a primary-expression within a constructor of a struct, it is classified as a variable. The type of the variable is the struct within which the reference occurs, and the variable represents the struct being constructed. The this variable of a constructor of a struct behaves exactly the same as an out parameter of the struct type-this in particular means that the variable must be definitely assigned in every execution path of the constructor.
- When this is used in a primary-expression within an instance method or instance accessor of a struct, it is classified as a variable. The type of the variable is the struct within which the reference occurs, and the variable represents the struct for which the method or accessor was invoked. The this variable of an instance method of a struct behaves exactly the same as a ref parameter of the struct type.

Use of this in a primary-expression in a context other than the ones listed above is an error. In particular, it is not possible to refer to this in a static method, a static property accessor, or in a variable-initializer of a field declaration.

Base access

A base-access consists of the reserved word base followed by either a "." token and an identifier or an expression-list enclosed in square brackets.

A base-access is used to access base class members that are hidden by similarly named members in the current class or struct. A base-access is permitted only in the block of a constructor, an instance method, or an instance accessor. When base.I occurs in a class or struct, I must denote

a member of the base class of that class or struct. Likewise, when base[E] occurs in a class, an applicable indexer must exist in the base class.

At compile-time, base-access expressions of the form base.I and base[E] are evaluated exactly as if they were written ((B)this).I and ((B)this)[E], where B is the base class of the class or struct in which the construct occurs. Thus, base.I and base[E] correspond to this.I and this[E], except this is viewed as an instance of the base class.

When a base-access references a function member (a method, property, or indexer), the function member is considered non-virtual for purposes of function member invocation. Thus, within an override of a virtual function member, a base-access can be used to invoke the inherited implementation of the function member. If the function member referenced by a base-access is abstract, an error occurs.

Postfix increment and decrement operators

The operand of a postfix increment or decrement operation must be an expression classified as a variable, a property access, or an indexer access. The result of the operation is a value of the same type as the operand.

If the operand of a postfix increment or decrement operation is a property or indexer access, the property or indexer must have both a get and a set accessor. If this is not the case, a compile-time error occurs.

Unary operator overload resolution is applied to select a specific operator implementation. Predefined ++ and -- operators exist for the following types: sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, and any enum type. The predefined ++ operators return the value produced by adding 1 to the operand, and the predefined -- operators return the value produced by subtracting 1 from the operand.

The run-time processing of a postfix increment or decrement operation of the form x++ or x-- consists of the following steps:

If x is classified as a variable:

- x is evaluated to produce the variable.
- The value of x is saved.
- The selected operator is invoked with the saved value of x as its argument.
- The value returned by the operator is stored in the location given by the evaluation of x.
- The saved value of x becomes the result of the operation.

If x is classified as a property or indexer access:

- The instance expression (if x is not static) and the argument list (if x is an indexer access) associated with x are evaluated, and the results are used in the subsequent get and set accessor invocations.
- The get accessor of x is invoked and the returned value is saved.
- The selected operator is invoked with the saved value of x as its argument.
- The set accessor of x is invoked with the value returned by the operator as its value argument.
- The saved value of x becomes the result of the operation.

The ++ and -- operators also support prefix notation. The result of x++ or x-- is the value of x before the operation, whereas the result of ++x or --x is the value of x after the operation. In either case, x itself has the same value after the operation.

An operator ++ or operator -- implementation can be invoked using either postfix or prefix notation. It is not possible to have separate operator implementations for the two notations.

New Operator

The new operator is used to create new instances of types. There are three forms of new expressions:

- Object creation expressions are used to create a new instances of class types and value types.
- Array creation expressions are used to create new instances of array types.
- Delegate creation expressions are used to create new instances of delegate types.

The new operator implies creation of an instance of a type, but does not necessarily imply dynamic allocation of memory. In particular, instances of value types require no additional memory beyond the variables in which they reside, and no dynamic allocations occur when new is used to create instances of value types.

Object creation expressions

An object-creation-expression is used to create a new instance of a class-type or a value-type. The type of an object-creation-expression must be a class-type or a value-type. The type cannot be an abstract class-type. The optional argument-list is permitted only if the type is a class-type or a struct-type.

Array creation expressions

An array-creation-expression is used to create a new instance of an array-type. An array creation expression of first form allocates an array instance of the type that results from deleting each of the individual expressions from the expression list. For example, the array creation expression new int[10, 20] produces an array instance of type int[,], and the array creation expression new int[10][,] produces an array of type int[][,]. Each expression in the expression list must be of type int, uint, long, or ulong, or of a type that can be implicitly converted to one or more of these types. The value of each expression determines the length of the corresponding dimension in the newly allocated array instance.

If an array creation expression of the first form includes an array initializer, each expression in the expression list must be a constant and the rank and dimension lengths specified by the expression list must match those of the array initializer.

In an array creation expression of the second form, the rank of the specified array type must match that of the array initializer. The individual dimension lengths are inferred from the number of elements in each of the corresponding nesting levels of the array initializer. Thus, the expression

```
new int[,] {{0, 1}, {2, 3}, {4, 5}};
```

exactly corresponds to

```
new int[3, 2] {{0, 1}, {2, 3}, {4, 5}};
```

The result of evaluating an array creation expression is classified as a value, namely a reference to the newly allocated array instance. The run-time processing of an array creation expression consists of the following steps:

- The dimension length expressions of the expression-list are evaluated in order, from left to right. Following evaluation of each expression, an implicit conversion to type int is performed. If evaluation of an expression or the subsequent implicit conversion causes an exception, then no further expressions are evaluated and no further steps are executed.
- The computed values for the dimension lengths are validated. If one or more of the values are less than zero, an IndexOutOfRangeException is thrown and no further steps are executed.
- An array instance with the given dimension lengths is allocated. If there is not enough memory available to allocate the new instance, an OutOfMemoryException is thrown and no further steps are executed.
- All elements of the new array instance are initialized to their default values.

- If the array creation expression contains an array initializer, then each expression in the array initializer is evaluated and assigned to its corresponding array element. The evaluations and assignments are performed in the order the expressions are written in the array initializer-in other words, elements are initialized in increasing index order, with the rightmost dimension increasing first. If evaluation of a given expression or the subsequent assignment to the corresponding array element causes an exception, then no further elements are initialized (and the remaining elements will thus have their default values).

An array creation expression permits instantiation of an array with elements of an array type, but the elements of such an array must be manually initialized. For example, the statement

```
int[][] a = new int[100][];
```

creates a single-dimensional array with 100 elements of type int[]. The initial value of each element is null. It is not possible for the same array creation expression to also instantiate the sub-arrays, and the statement

```
int[][] a = new int[100][5];              // Error
```

is an error. Instantiation of the sub-arrays must instead be performed manually, as in

```
int[][] a = new int[100][];
for (int i = 0; i < 100; i++) a[i] = new int[5];
```

When an array of arrays has a "rectangular" shape, that is when the sub-arrays are all of the same length, it is more efficient to use a multi-dimensional array. In the example above, instantiation of the array of arrays creates 101 objects-one outer array and 100 sub-arrays. In contrast,

```
int[,] = new int[100, 5];
```

creates only a single object, a two-dimensional array, and accomplishes the allocation in a single statement.

Delegate creation expressions

A delegate-creation-expression is used to create a new instance of a delegate-type. The argument of a delegate creation expression must be a method group or a value of a delegate-type. If the argument is a method group, it identifies the method and, for an instance method, the object for which to create a delegate. If the argument is a value of a delegate-type, it identifies a delegate instance of which to create a copy.

The compile-time processing of a delegate-creation-expression of the form new D(E), where D is a delegate-type and E is an expression, consists of the following steps:

If E is a method group:

- If the method group resulted from a base-access, an error occurs.
- The set of methods identified by E must include exactly one method with precisely the same signature and return type as those of D, and this becomes the method to which the newly created delegate refers. If no matching method exists, or if more than one matching methods exists, an error occurs. If the selected method is an instance method, the instance expression associated with E determines the target object of the delegate.
- As in a method invocation, the selected method must be compatible with the context of the method group: If the method is a static method, the method group must have resulted from a simple-name or a member-access through a type. If the method is an instance method, the method group must have resulted from a simple-name or a member-access through a variable or value. If the selected method does not match the context of the method group, an error occurs.
- The result is a value of type D, namely a newly created delegate that refers to the selected method and target object.

Otherwise, if E is a value of a delegate-type:

- The delegate-type of E must have the exact same signature and return type as D, or otherwise an error occurs.

- The result is a value of type D, namely a newly created delegate that refers to the same method and target object as E.
- Otherwise, the delegate creation expression is invalid, and an error occurs.

The run-time processing of a delegate-creation-expression of the form new D(E), where D is a delegate-type and E is an expression, consists of the following steps:

If E is a method group:

- If the method selected at compile-time is a static method, the target object of the delegate is null. Otherwise, the selected method is an instance method, and the target object of the delegate is determined from the instance expression associated with E:
- The instance expression is evaluated. If this evaluation causes an exception, no further steps are executed.
- If the instance expression is of a reference-type, the value computed by the instance expression becomes the target object. If the target object is null, a NullReferenceException is thrown and no further steps are executed.
- If the instance expression is of a value-type, a boxing operation is performed to convert the value to an object, and this object becomes the target object.
- A new instance of the delegate type D is allocated. If there is not enough memory available to allocate the new instance, an OutOfMemoryException is thrown and no further steps are executed.
- The new delegate instance is initialized with a reference to the method that was determined at compile-time and a reference to the target object computed above.

If E is a value of a delegate-type:

- E is evaluated. If this evaluation causes an exception, no further steps are executed.
- If the value of E is null, a NullReferenceException is thrown and no further steps are executed.
- A new instance of the delegate type D is allocated. If there is not enough memory available to allocate the new instance, an OutOfMemoryException is thrown and no further steps are executed.
- The new delegate instance is initialized with references to the same method and object as the delegate instance given by E.

The method and object to which a delegate refers are determined when the delegate is instantiated and then remain constant for the entire lifetime of the delegate. In other words, it is not possible to change the target method or object of a delegate once it has been created.

It is not possible to create a delegate that refers to a constructor, property, indexer, or user-defined operator.

As described above, when a delegate is created from a method group, the signature and return type of the delegate determine which of the overloaded methods to select. Consider the following code.

```
delegate double DoubleFunc(double x);
class One
{
  DoubleFunc f = new DoubleFunc(Square);
static float Square(float x)
{
  return x * x;
}
static double Square(double x)
{
```

```
    return x * x;
  }
  }
```

Here, the One.f field is initialized with a delegate that refers to the second Square method because that method exactly matches the signature and return type of DoubleFunc. Had the second Square method not been present, a compile-time error would have occurred.

Type of Operator

The typeof operator is used to obtain the System.Type object for a type. The result of a typeof-expression is the System.Type object for the indicated type.

Consider the following code.

```
  class TestClass
  {
    static void Main()
  {
    Type[] t = {
           typeof(int),
           typeof(System.Int32),
           typeof(string),
           typeof(double[])
           };
      for (int i = 0; i < t.Length; i++)
  {
      Console.WriteLine(t[i].Name);
    }
  }
  }
```

It produces the following output:

```
Int32
Int32
String
Double[]
```

Note that int and System.Int32 are the same type.

checked and unchecked operators

The checked and unchecked operators are used to control the overflow checking context for integral-type arithmetic operations and conversions.

The checked operator evaluates the contained expression in a checked context, and the unchecked operator evaluates the contained expression in an unchecked context. A checked-expression or unchecked-expression corresponds exactly to a parenthesized-expression, except that the contained expression is evaluated in the given overflow checking context.

The overflow checking context can also be controlled through the checked and unchecked statements. The following operations are affected by the overflow checking context established by the checked and unchecked operators and statements:

- The predefined ++ and -- unary operators, when the operand is of an integral type.
- The predefined - unary operator, when the operand is of an integral type.
- The predefined +, -, *, and / binary operators, when both operands are of integral types.
- Explicit numeric conversions from one integral type to another integral type.

When one of the above operations produce a result that is too large to represent in the destination type, the context in which the operation is performed controls the resulting behavior:

- In a checked context, if the operation is a constant expression, a compile-time error occurs. Otherwise, when the operation is performed at run-time, an OverflowException is thrown.
- In an unchecked context, the result is truncated by discarding any high-order bits that do not fit in the destination type.

When a non-constant expression (an expression that is evaluated at run-time) is not enclosed by any checked or unchecked operators or statements, the effect of an overflow during the run-time evaluation of the expression depends on external factors (such as compiler switches and execution environment configuration). The effect is however guaranteed to be either that of a checked evaluation or that of an unchecked evaluation.

For constant expressions (expressions that can be fully evaluated at compile-time), the default overflow checking context is always checked. Unless a constant expression is explicitly placed in an unchecked context, overflows that occur during the compile-time evaluation of the expression always cause compile-time errors.

Consider the following code.

```
class TestClass
{
  static int x = 1000000;
  static int y = 1000000;
static int FunctionOne()
{
  return checked(x * y);    // Throws OverflowException
}
static int FunctionTwo()
{
  return unchecked(x * y); // Returns -727379968
}
static int FunctionThree()
{

return x * y;      // Depends on default
}
}
```

Here, no compile-time errors are reported since neither of the expressions can be evaluated at compile-time. At run-time, the FunctionOne() method throws an OverflowException, and the FunctionThree() method returns -727379968 (the lower 32 bits of the out-of-range result). The behavior of the FunctionThree() method depends on the default overflow checking context for the compilation, but it is either the same as FunctionOne() or the same as FunctionTwo().

Consider the following code.

```
class TestClass
{
  const int x = 1000000;
  const int y = 1000000;
  static int FunctionOne()
{
     return checked(x * y);    // Compile error, overflow
  }
  static int FunctionTwo()
{
```

```
    return unchecked(x * y); // Returns -727379968
  }
  static int FunctionThree()
{
    return x * y;      // Compile error, overflow
  }
}
```

Here, the overflows that occur when evaluating the constant expressions in FunctionOne() and FunctionThree() cause compile-time errors to be reported because the expressions are evaluated in a checked context. An overflow also occurs when evaluating the constant expression in FunctionTwo(), but since the evaluation takes place in an unchecked context, the overflow is not reported.

The checked and unchecked operators only affect the overflow checking context for those operations that are textually contained within the "(" and ")" tokens. The operators have no effect on function members that are invoked as a result of evaluating the contained expression. Consider the following code.

```
class TestClass
{
  static int Multiply(int x, int y)
{
    return x * y;
  }
static int FunctionOne()
{
  return checked(Multiply(1000000, 1000000));
}
}
```

Here, the use of checked in FunctionOne() does not affect the evaluation of x * y in Multiply(), and x * y is therefore evaluated in the default overflow checking context.

The unchecked operator is convenient when writing constants of the signed integral types in hexadecimal notation. Consider the following code.

```
class TestClass
{
  public const int AllBits = unchecked((int)0xFFFFFFFF);
  public const int HighBit = unchecked((int)0x80000000);
}
```

Both of the hexadecimal constants above are of type uint. Because the constants are outside the int range, without the unchecked operator, the casts to int would produce compile-time errors.

Function members

Function members are members that contain executable statements. Function members are always members of types and cannot be members of namespaces. C# defines the following five categories of function members:

- Methods
- Constructors
- Properties
- Indexers and User-defined operators

The statements contained in function members are executed through function member invocations. The actual syntax for writing a function member invocation depends on the particular function

member category. However, all function member invocations are expressions, allow arguments to be passed to the function member, and allow the function member to compute and return a result.

The argument list of a function member invocation provides actual values or variable references for the parameters of the function member.

Invocations of constructors, methods, indexers, and operators employ overload resolution to determine which of a candidate set of function members to invoke.

Argument lists

Every function member invocation includes an argument list which provides actual values or variable references for the parameters of the function member. The syntax for specifying the argument list of a function member invocation depends on the function member category:

- For constructors, methods, and delegates, the arguments are specified as an argument-list, as described below.
- For properties, the argument list is empty when invoking the get accessor, and consists of the expression specified as the right operand of the assignment operator when invoking the set accessor.
- For indexers, the argument list consists of the expressions specified between the square brackets in the indexer access. When invoking the set accessor, the argument list additionally includes the expression specified as the right operand of the assignment operator.
- For user-defined operators, the argument list consists of the single operand of the unary operator or the two operands of the binary operator.

The arguments of properties, indexers, and user-defined operators are always passed as value parameters. Reference and output parameters are not supported for these categories of function members.

The arguments of a constructor, method, or delegate invocation are specified as an argument-list. An argument-list consists of zero or more arguments, separated by commas. Each argument can take one of the following forms:

- An expression, indicating that the argument is passed as a value parameter.
- The keyword ref followed by a variable-reference, indicating that the argument is passed as a reference parameter. A variable must be definitely assigned before it can be passed as a reference parameter.
- The keyword out followed by a variable-reference, indicating that the argument is passed as an output parameter. A variable is considered definitely assigned following a function member invocation in which the variable is passed as an output parameter.

Methods, indexers, and constructors may declare their last parameter to be a parameter array. Such function members are invoked either in their normal form or in their expanded form depending on which is applicable:

- When a function member with a parameter array is invoked in its normal form, the argument given for the parameter array must be a single expression of a type that is implicitly convertible to the parameter array type. In this case, the parameter array acts precisely like a value parameter.
- When a function member with a parameter array is invoked in its expanded form, the invocation must specify zero or more arguments for the parameter array, where each argument is an expression of a type that is implicitly convertible to the element type of the parameter array. In this case, the invocation creates an instance of the parameter array type with a length corresponding to the number of arguments, initializes the elements of the array instance with the given argument values, and uses the newly created array instance as the actual argument.

The expressions of an argument list are always evaluated in the order they are written. Consider the following code.

```
class TestClass
{
  static void FunctionOne(int x, int y, int z)
{
    Console.WriteLine("x = {0}, y = {1}, z = {2}", x, y, z);
  }
static void Main()
{
  int i = 0;
  FunctionOne(i++, i++, i++);
}
}
```

It produces the output

x = 0, y = 1, z = 2

Overload resolution

Overload resolution is a mechanism for selecting the best function member to invoke given an argument list and a set of candidate function members. Overload resolution selects the function member to invoke in the following distinct contexts within C#:

- Invocation of a method named in an invocation-expression.
- Invocation of a constructor named in an object-creation-expression.
- Invocation of an indexer accessor through an element-access.
- For each argument in A, the parameter passing mode of the argument is identical to the parameter passing mode of the corresponding parameter, and
- for a value parameter or a parameter array, an implicit conversion exists from the type of the argument to the type of the corresponding parameter, or
- for a ref or out parameter, the type of the argument is identical to the type of the corresponding parameter.

Function member invocation

This section describes the process that takes place at run-time to invoke a particular function member. It is assumed that a compile-time process has already determined the particular member to invoke, possibly by applying overload resolution to a set of candidate function members.

For purposes of describing the invocation process, function members are divided into two categories:

- Static function members. These are static methods, constructors, static property accessors, and user-defined operators. Static function members are always non-virtual.
- Instance function members. These are instance methods, instance property accessors, and indexer accessors. Instance function members are either non-virtual or virtual, and are always invoked on a particular instance. The instance is computed by an instance expression, and it becomes accessible within the function member as this.

Student Activity 4

1. Explain the concept of various type of constructor with the help of example.
2. What is a destructor?
3. Explain object creation expressions. Can the type of object creation expression be an abstract class-type?

4. What do you understand by argument list? What are the forms that an argument can take?
5. What do you understand by overload resolution?

4.9 DELEGATES

Delegates enable scenarios that other languages-C++, Pascal, Modula, and others-have addressed with function pointers. Unlike C++ function pointers, delegates are fully object oriented; unlike C++ pointers to member functions, delegates encapsulate both an object instance and a method.

A delegate declaration defines a class that extends the class System.Delegate. A delegate instance encapsulates a method-a callable entity. For instance methods, a callable entity consists of an instance and a method on the instance. For static methods, a callable entity consists of just a method. If you have a delegate instance and an appropriate set of arguments, you can invoke the delegate with the arguments.

An interesting and useful property of a delegate is that it does not know or care about the class of the object that it references. Any object will do; all that matters is that the method's signature matches the delegate's. This makes delegates perfectly suited for "anonymous" invocation.

Delegate declarations

A delegate-declaration is a type-declaration that declares a new delegate type. It is an error for the same modifier to appear multiple times in a delegate declaration. The new modifier is only permitted on delegates declared within another type. It specifies that the delegate hides an inherited member by the same name.

The public, protected, internal, and private modifiers control the accessibility of the delegate type. Depending on the context in which the delegate declaration occurs, some of these modifiers may not be permitted.

The formal-parameter-list identifies the signature of the delegate, and the result-type indicates the return type of the delegate. The signature and return type of the delegate must exactly match the signature and return type of any method that the delegate type encapsulates. Delegate types in C# are name equivalent, not structurally equivalent. Two different delegates types that have the same signature and return type are considered different delegate types.

A delegate type is a class type that is derived from System.Delegate. Delegate types are implicitly sealed: it is not permissible to derive any type from a delegate type. It is also not permissible to derive a non-delegate class type from System.Delegate. Note that System.Delegate is not itself a delegate type, it is a class type that all delegate types derive from.

C# provides special syntax for delegate instantiation and invocation. Except for instantiation, any operation that can be applied to a class or class instance can also be applied to a delegate class or instance. In particular, it is possible to access members of the System.Delegate type via the usual member access syntax.

Combinable delegate types

Delegate types are classified into two kinds: combinable and non-combinable. A combinable delegate type must satisfy the following conditions:

- The declared return type of the delegate must be void.
- None of the parameters of the delegate type can be declared as output parameters.

A run-time exception occurs if an attempt is made to combine two instances of a non-combinable delegate types unless one or the other is null.

Delegate instantiation

Although delegates behave in most ways like other classes, C# provides special syntax for instantiating a delegate instance. A delegate-creation-expression is used to create a new instance of a delegate. The newly created delegate instance then refers to either:

- The static method referenced in the delegate-creation-expression, or
- The target object (which cannot be null) and instance method referenced in the delegate-creation-expression, or

Another delegate

Once instantiated, delegate instances always refer to the same target object and method.

Multi-cast delegates

Delegates can be combined using the addition operator, and one delegate can be removed from another using the subtraction operator. A delegate instance created by combining two or more (non-null) delegate instances is called a multicast delegate instance. For any delegate instance, the invocation list of the delegate is defined as the ordered list of non-multicast delegates that would be invoked if the delegate instance were invoked. More precisely:

- For a non-multicast delegate instance, the invocation list consists of the delegate instance itself.
- For a multi-cast delegate instance that was created by combining two delegates, the invocation list is the formed by concatenating the invocation lists of the two operands of the addition operation that formed the multi-cast delegate.

Delegate invocation

C# provides special syntax for invoking a delegate. When a non-multicast delegate is invoked, it invokes the method that the delegate refers to with the same arguments, and returns the same value that the referred to method returns. If an exception occurs during the invocation of a delegate, and the exception is not caught within the method that was invoked, the search for an exception catch clause continues in the method that called the delegate, as if that method had directly called the method that the delegate referred to.

Invocation of a multi-cast delegate proceeds by invocation each of the delegates on the invocation list, in order. Each call is passed the same set of arguments. If the delegate includes reference parameters, each method invocation will occur with a reference to the same variable; changes to that variable by one method in the invocation list will be "seen" by any later methods in the invocation list.

If an exception occurs during processing of the invocation of a multicast delegate, and the exception is not caught within the method that was invoked, the search for an exception catch clause continues in the method that called the delegate, and any methods later in the invocation list are not invoked.

Boxing and unboxing

Boxing and unboxing is a central concept in C#'s type system. It provides a binding link between value-types and reference-types by permitting any value of a value-type to be converted to and from type object. Boxing and unboxing enables a unified view of the type system wherein a value of any type can ultimately be treated as an object.

Boxing conversions

A boxing conversion permits any value-type to be implicitly converted to the type object or to any interface-type implemented by the value-type. Boxing a value of a value-type consists of allocating an object instance and copying the value-type value into that instance.

The actual process of boxing a value of a value-type is best explained by imagining the existence of a boxing class for that type. For any value-type T, the boxing class would be declared as follows:

```
class T_Box
{
  T value;
  T_Box(T t) {
    value = t;
  }
}
```

Boxing of a value v of type T now consists of executing the expression new T_Box(v), and returning the resulting instance as a value of type object. Thus, the statements

```
int i = 123;
object box = i;
```

conceptually correspond to

```
int i = 123;
object box = new int_Box(i);
```

Boxing classes like T_Box and int_Box above don't actually exist and the dynamic type of a boxed value isn't actually a class type. Instead, a boxed value of type T has the dynamic type T, and a dynamic type check using the is operator can simply reference type T. For example,

```
int i = 123;
object box = i;
if (box is int) {
  Console.Write("Box contains an int");
}
```

will output the string "Box contains an int" on the console.

A boxing conversion implies making a copy of the value being boxed. This is different from a conversion of a reference-type to type object, in which the value continues to reference the same instance and simply is regarded as the less derived type object. For example, given the declaration

```
struct Point
{
  public int x, y;
  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }
}
```

the following statements

```
Point p = new Point(10, 10);
object box = p;
p.x = 20;
Console.Write(((Point)box).x);
```

will output the value 10 on the console because the implicit boxing operation that occurs in the assignment of p to box causes the value of p to be copied. Had Point instead been declared a class, the value 20 would be output because p and box would reference the same instance.

Unboxing conversions

An unboxing conversion permits an explicit conversion from type object to any value-type or from any interface-type to any value-type that implements the interface-type. An unboxing

operation consists of first checking that the object instance is a boxed value of the given value-type, and then copying the value out of the instance.

Referring to the imaginary boxing class described in the previous section, an unboxing conversion of an object box to a value-type T consists of executing the expression ((T_Box)box).value. Thus, the statements

```
object box = 123;
int i = (int)box;
```

conceptually correspond to

```
object box = new int_Box(123);
int i = ((int_Box)box).value;
```

For an unboxing conversion to a given value-type to succeed at run-time, the value of the source argument must be a reference to an object that was previously created by boxing a value of that value-type. If the source argument is null or a reference to an incompatible object, an InvalidCastException is thrown.

Hiding

The scope of an entity typically encompasses more program text than the declaration space of the entity. In particular, the scope of an entity may include declarations that introduce new declaration spaces containing entities of the same name. Such declarations cause the original entity to become hidden. Conversely, an entity is said to be visible when it is not hidden.

Name hiding occurs when scopes overlap through nesting and when scopes overlap through inheritance. The characteristics of the two types of hiding are described in the following sections.

Hiding through nesting

Name hiding through nesting can occur as a result of nesting namespaces or types within namespaces, as a result of nesting types within classes or structs, and as a result of parameter and local variable declarations. Name hiding through nesting of scopes always occurs "silently", i.e. no errors or warnings are reported when outer names are hidden by inner names.

In the example

```
class A
{

  int i = 0;
    void F()  { int i = 1;}
    void G()  {i=1;}
}
```

within the F method, the instance variable i is hidden by the local variable i, but within the G method, i still refers to the instance variable.

When a name in an inner scope hides a name in an outer scope, it hides all overloaded occurrences of that name. In the example

```
class Outer
{
     static void F(int i) {}
     static void F(string s) {}
     class Inner {
     void G() {
     F(1);     // Invokes Outer.Inner.F
     F("Hello"); // Error
   }
     static void F(long l) {}
   }}
```

the call F(1) invokes the F declared in Inner because all outer occurrences of F are hidden by the inner declaration. For the same reason, the call F("Hello") is in error.

Hiding through inheritance

Name hiding through inheritance occurs when classes or structs redeclare names that were inherited from base classes. This type of name hiding takes one of the following forms:

A constant, field, property, event, or type introduced in a class or struct hides all base class members with the same name.

A method introduced in a class or struct hides all non-method base class members with the same name, and all base class methods with the same signature (method name and parameter count, modifiers, and types).

An indexer introduced in a class or struct hides all base class indexers with the same signature (parameter count and types).

The rules governing operator declarations make it impossible for a derived class to declare an operator with the same signature as an operator in a base class. Thus, operators never hide one another.

Contrary to hiding a name from an outer scope, hiding an accessible name from an inherited scope causes a warning to be reported. In the example

```
class Base
{
  public void F() {}
}
class Derived: Base
{
  public void F() {} // Warning, hiding an inherited name
}
```

the declaration of F in Derived causes a warning to be reported. Hiding an inherited name is specifically not an error, since that would preclude separate evolution of base classes. For example, the above situation might have come about because a later version of Base introduced a F method that wasn't present in an earlier version of the class. Had the above situation been an error, then any change made to a base class in a separately versioned class library could potentially cause derived classes to become invalid.

The warning caused by hiding an inherited name can be eliminated through use of the new modifier:

```
class Base
{
  public void F() {}
}
class Derived: Base
{
  new public void F() {}
}
```

The new modifier indicates that the F in Derived is "new", and that it is indeed intended to hide the inherited member.

A declaration of a new member hides an inherited member only within the scope of the new member.

```
class Base
{
  public static void F() {}
```

```
}
class Derived: Base
{
  new private static void F() {}// Hides Base.F in Derived only
}
class MoreDerived: Derived
{
  static void G() { F(); }    // Invokes Base.F
}
```

In the example above, the declaration of F in Derived hides the F that was inherited from Base, but since the new F in Derived has private access, its scope does not extend to MoreDerived. Thus, the call F() in MoreDerived.G is valid and will invoke Base.F.

A derived class can hide inherited members by declaring new members with the same name or signature. Note however that hiding an inherited member does not remove the member-it merely makes the member inaccessible in the derived class.

The intuitive rule for hiding in multiple-inheritance interfaces is simply this: If a member is hidden in any access path, it is hidden in all access paths.

Student Activity 5

1. What is a delegate? What are the types of delegates?
2. Is this feasible to multicast a delegate? Justify your answer.
3. Explain boxing and unboxing conversions.
4. What do you understand by hiding? Explain hiding through nesting and hiding through inheritance.

4.10 SUMMARY

- An abstract class cannot instantiate directly.
- Class A is said to be the direct base class of B if B is inherited from A.
- A static method does not operate on a specific instance, and it is an error to refer to this in a static method.
- When an instance method declaration includes an override modifier, the method is said to be an override method.
- An abstract method is implicitly also a virtual method.
- A property that includes both a get accessor and a set accessor is said to be read -write property.
- A property can be a static member whereas an idexer is always an instance member.
- A delegates declaration defines a class that extends the class System.Delegates.
- Delegates types are classified into two kinds: combinable and non-combinable.
- Once instantiated, delegates instances always refer to the same target object and method.

4.11 KEYWORDS

- **Object:** Objects are the basic runtime entity in an object-oriented system.
- **Class:** A set of related objects is called a class
- **Inheritance:** The ability of a class to inherit the properties of another class is called inheritance.
- **Polymorphism:** Capability of the data to be processed in more than one form is called polymorphism.

- **Field:** A field is a member that represent a variable associated with an object or class.
- **Property:** A property is a member that provides access to an attribute of an object or a class.
- **Event:** An event is a member that enables an object or class to provide notification.
- **Indexer:** An indexer is a member that enables an object to be indexed in the same way as an array.
- **Dynamic Binding:** It refers to the linking of a procedure call to the code to be executed in response to the call.
- **Data Abstraction:** It refers to the act of representing essential features without including the background details or explanations.

4.12 REVIEW QUESTIONS

1. What is the role of a constructor method in a class?
2. What is meant by method overloading? Give some examples.
3. How is a static member different from other members?
4. What are copy constructors? Where are they used?
5. What are indexers? How are they useful?
6. Can you allow class to be inherited, but prevent the method from being over-ridden?
7. Declare a class having some methods and show how these methods may be invoked.
8. What will be the output of the following code?

```
class Test
{
  static void F(params object[] args)
  {
    foreach (object o in a)
    {
      Console.Write(o.GetType().FullName);
      Console.Write(" ");
    }
    Console.WriteLine();
  }
  static void Main()
  {
    object[] a = {1, "Delhi", 93.1};
    object o = a;
    F(a);
    F((object)a);
    F(o);
    F((object[])o);
  }
}
```

9. State one usage each of different forms of Main method.
10. How is a value type parameter different from a reference type parameter?
11. Explain the meaning of method overriding.
12. Wap to calculate the factorial of a number using recursion.
13. WAP to calculate the sum of array when array is passed as a parameter.

4.13 FURTHER READINGS

CLR via C#, Second Edition (Paperback) by Jeffrey Richter, Publisher: Microsoft Press; 2nd edition (February 22, 2006)

Essential C# 2.0 (Microsoft .NET Development Series) (Paperback) by Mark Michaelis, Publisher: Addison-Wesley Professional (July 13, 2006)

C# Programming Language, The (2nd Edition) (Microsoft .NET Development Series) (Hardcover) by Anders Hejlsberg, Scott Wiltamuth, Peter Golde, Publisher: Addison-Wesley Professional; 2 edition (June 9, 2006)

Programming in the Key of C# (Paperback) by Charles Petzold, Publisher: Microsoft Press; 1 edition (July 30, 2003)

UNIT

5

CONTROL STATEMENTS

L E A R N I N G O B J E C T I V E S

After studying this unit, you should be able to

- Define statements and blocks
- Describe branching and switch statements
- Understand about the foreach statement
- Describe lock statement

U N I T S T R U C T U R E

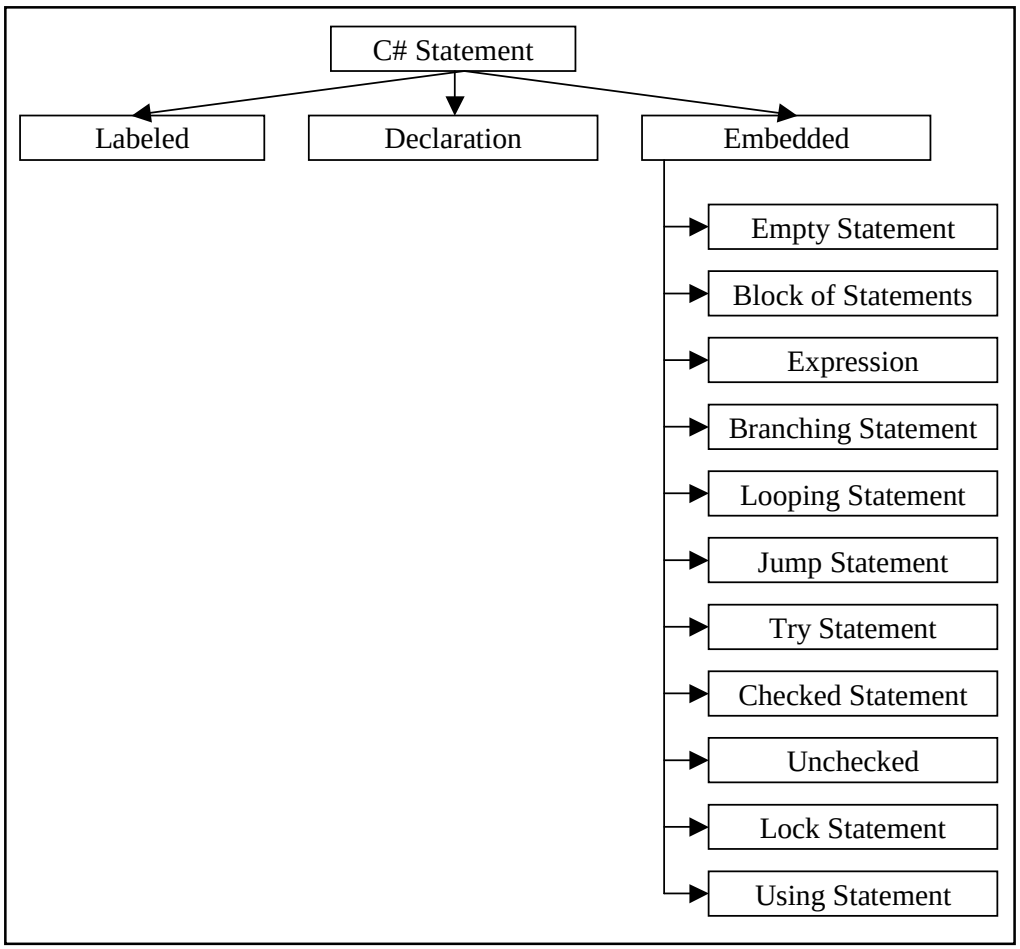
5.1 INTRODUCTION

This chapter will introduce you to the various control statements in C#, blocks, statement lists, the various looping statements such as the while statement, the do statement, for statement etc. in which the looping allows repeated execution of statements. Also you will be introduced to jump statements, break statement, go to statements and lastly the checked and unchecked statements.

5.2 STATEMENTS

In the context of a programming language a complete sentence made up of reserved and user-defined words is called a statement. Statements serve a variety of purposes. Some statements indicate execution actions while some simply direct the compilation process. Just as a complete story consists of a number of coherent sentences, a complete program is made up of a number of coherent statements.

C# provides a rich variety of statements that includes all the familiar statements of languages C and C++. The types of statements available in C# are presented in the following tree diagram.



Labeled Statement

A labeled statement is any valid statement that is identifiable by a user-defined label attached to it. The syntax of a labeled-statement is given below.

identifier: statement

For example, consider the following C# function.

```
int abc(int x)
{
  if (x >= 0) goto doit;
  x = -x;
  doit: return x;
}
```

In the code-snippet listed above, identifier - doit - is a label and hence - doit: return x; - is a labeled-statement.

A label can be referenced from goto statements within the scope of the label. This means that goto statements can transfer control inside blocks and out of blocks, but never into blocks.

A labeled-statement permits a statement to be prefixed by a label. Labeled statements are permitted blocks, but are not permitted as embedded statements.

A labeled statement declares a label with the name given by an identifier. The scope of a label is the block in which the label is declared, including any nested blocks. It is an error for two labels with the same name to have overlapping scopes.

Labels have their own declaration space and do not interfere with other identifiers. Consider the following code.

```
int FunctionOne(int x)
{
  if (x >= 0) goto x;
  x = -x;
  x: return x;
}
```

The label name - x- is valid and uses the name x as both a parameter and a label.

Execution of a labeled statement corresponds exactly to execution of the statement following the label. In addition to the reachability provided by normal flow of control, a labeled statement is reachable if the label is referenced by a reachable goto statement.

End points and reachability

A statement starts at a point and ends at another point. The end point of a statement is the location that immediately follows the statement. The execution rules for composite statements (statements that contain embedded statements) specify the action that is taken when control reaches the end point of an embedded statement. For example, when control reaches the end point of a statement in a block, control is transferred to the next statement in the block.

If a statement can possibly be reached by execution, the statement is said to be reachable. Conversely, if there is no possibility that a statement will be executed, the statement is said to be unreachable.

Consider the following code.

```
void FunctionOne()
{
  Console.WriteLine("This is reachable");
  goto Label;
  Console.WriteLine("This is unreachable");
  Label:
    Console.WriteLine("reachable through label");
}
```

Here, the second Console.WriteLine invocation is unreachable because there is no possibility that the statement will be executed. A warning is reported if the compiler determines that a statement is unreachable. However, it is specifically not an error for a statement to be unreachable.

The compiler performs flow analysis according to the reachability rules defined for each statement to determine whether a particular statement or end point. The flow analysis takes into account the values of constant expressions that control the behavior of statements, but the possible values of non-constant expressions are not considered. In other words, for purposes of control flow analysis, a non-constant expression of a given type is considered to have any possible value of that type.

Consider the following code.

```
void FunctionOne()
{
  const int i = 1;
  if (i == 2) Console.WriteLine("This is unreachable");
}
```

Here, the boolean expression of the if statement is a constant expression because both operands of the == operator are constants. The constant expression is evaluated at compile-time, producing the value false, and the Console.WriteLine invocation is therefore considered unreachable. However, if i is changed to be a local variable as is listed below the statement becomes reachable.

```
void FunctionOne()
{
  int i = 1;
  if (i == 2) Console.WriteLine("This is reachable");
}
```

The Console.WriteLine invocation is considered reachable, even though it will in reality never be executed. The block of a function member is always considered reachable. By successively evaluating the reachability rules of each statement in a block, the reachability of any given statement can be determined. Consider the following code.

```
void FunctionOne(int x)
{
  Console.WriteLine("start");
  if (x < 0) Console.WriteLine("negative");
}
```

In the above example, the reachability of the second Console.WriteLine is determined as follows:

- First, because the block of the FunctionOne method is reachable, the first Console.WriteLine statement is reachable.
- Next, because the first Console.WriteLine statement is reachable, its end point is reachable.
- Next, because the end point of the first Console.WriteLine statement is reachable, the if statement is reachable.
- Finally, because the boolean expression of the if statement does not have the constant value false, the second Console.WriteLine statement is reachable.

Note that there are two situations in which it is an error for the end point of a statement to be reachable:

- Because the switch statement does not permit a switch section to "fall through" to the next switch section, it is an error for the end point of the statement list of a switch section to be reachable. If this error occurs, it is typically an indication that a break statement is missing.
- It is an error for the end point of the block of a function member that computes a value to be reachable. If this error occurs, it is typically an indication that a return statement is missing.

5.3 BLOCKS

Multiple statements can be blocked to make them behave as a single statement. A block consists of an optional statement-list, enclosed in braces. If the statement list is omitted, the block is said to be empty. A block may contain declaration statements. The scope of a local variable or constant declared in a block extends from the declaration to the end of the block. Within a block, the meaning of a name used in an expression context must always be the same.

Consider the following code.

```
Int FunctionOne(int x)
{
  if(x < 10)
    return(2 * x);
  else
    {
      System.Console.WriteLine("Blocked statements");
      Return(x);
    }
}
```

Here the two statements in the else part form a single block of statements. It is treated as a single statement in the context of else.

A block is executed as follows:

- If the block is empty, control is transferred to the end point of the block.
- If the block is not empty, control is transferred to the statement list. When and if control reaches the end point of the statement list, control is transferred to the end point of the block.

The statement list of a block is reachable if the block itself is reachable. The end point of a block is reachable if the block is empty or if the end point of the statement list is reachable.

A statement list consists of one or more statements written in sequence. Statement lists occur in blocks and in switch-blocks.

A statement list is executed by transferring control to the first statement. When and if control reaches the end point of a statement, control is transferred to the next statement. When and if control reaches the end point of the last statement, control is transferred to the end point of the statement list.

A statement in a statement list is reachable if at least one of the following is true:

- The statement is the first statement and the statement list itself is reachable.
- The end point of the preceding statement is reachable.
- The statement is a labeled statement and the label is referenced by a reachable goto statement.

The end point of a statement list is reachable if the end point of the last statement in the list is reachable.

A block of statements having no statement is called an empty-statement. It is a do-nothing statement. Execution of an empty statement simply transfers control to the end point of the statement. Thus, the end point of an empty statement is reachable if the empty statement is reachable.

Branching statements

Execution of a group of statements takes place in sequential manner in their order of appearance. However, often it is required that based on some conditional criteria a specified statement is executed while other are omitted. Jumping to a statement depending on some selection basis is termed as branching.

C# offers two broad mechanism to accomplish branching in a program.

```
if-statement
switch-statement
The - if - statement
```

Working of an - if - statement is straightforward. The general syntax of the - if - statement is given below.

```
   If (condition)
statement1;
   else
     statement2;
```

A condition is an expression or a statement that evaluates to either of the two Boolean values - true or false.

Note that the indentation shown above is not mandatory. It has been used on purpose. It enhances the readability of the code. However, nobody stops you from writing it as any one of the forms given below.

```
   If (condition) statement1; else statement2;
   Or
   If (condition) statement1;
   else statement2;
   Or
   If (condition) statement1;
   Or
   If (condition)
;
   else
     statement2;
```

In all the cases what the - if - statement does is described in the following points.

1. The condition is evaluated
2. If the condition is found to be true then statement1 is executed and if it turns out to be false then statement2 is executed
3. Either of the statements - statement1 and statement2 - may be missing in which case no action takes place in that portion of the - if - statement.
4. The statements - statement1 and statement2 - may be a single statement or a block of statements. Multiple statements must be blocked in the else part otherwise only the first statement shall be takes to be associated with the else part. Consider the following code snippet.

```
if(condition)
  statement1;
else
  statement2;
       statement3;
```

The indentation of the above statements suggests that if the condition is false both the statements - statement1 and statement2 - should be executed. However, as far as C# is concerned it associates only statement2 to the else part. Therefore, irrespective of the value of the condition, statement3 will always get executed. If you intend to treat both the statements - statement1 and statement2 - in the else part you must block them equivalent to a single statement as shown below.

```
if(condition)
  statement1;
else
  {
statement2;
       statement3;
     }
```

5. The - else - portion of the s - if - statement is associated with the nearest preceding - if - that does not have its own - else. Consider the following code snippet.

```
if(condition1)
  if(condition2)
    statement1;
else
statement2;
```

The indentation in the above code seems to suggest that the - else - part should be associated with the - if - with condition1. However, since - else - is associated with the nearest preceding - if - without an - else -, actually - else - here is associated with conditio2.

The switch statement

An if-else statement essentially performs a two-way branching. It is suitable in cases where one of the two given statements is to be selected for execution. Though, it can also implement multi-way branching, the code becomes very involved. Take a look at the following code snippet.

```
    if(condition1)  statement1;
    else
if(condition2)  statement2;
     else
if(condition3)  statement3;
       else
if(condition4)  statement4;
         else
if(condition5)  statement5;
```

As the number of conditional statement increases the code becomes difficult to read and maintain.

The switch statement provides a more elegant way to handle multi-way branching. executes the statements that are associated with the value of the controlling expression. The syntax is given below.

```
switch (switch_expression)
{
case Switch_Value1:
       FunctionOne();
       break;
case Switch_Value2:
       FunctionTwo();
       break;
case Switch_Value3:
       FunctionThree();
       break;
default:
       FunctionOther();
       break;
}
```

A switch-statement consists of the keyword switch, followed by a parenthesized expression (called the switch expression), followed by a switch-block. The switch-block consists of zero or more switch-sections, enclosed in braces. Each switch-section consists of one or more switch-labels followed by a statement-list.

The governing type of a switch statement is established by the switch expression. If the type of the switch expression is sbyte, byte, short, ushort, int, uint, long, ulong, char, string, or an enum-type, then that is the governing type of the switch statement. Otherwise, exactly one user-defined implicit conversion must exist from the type of the switch expression to one of the following possible governing types: sbyte, byte, short, ushort, int, uint, long, ulong, char, string. If no such implicit conversion exists, or if more that one such implicit conversion exists, a compile-time error occurs.

The constant expression of each case label must denote a value of a type that is implicitly convertible to the governing type of the switch statement. A compile-time error occurs if two or more case labels in the same switch statement specify the same constant value.

There can be at most one default label in a switch statement.

A switch statement is executed as follows:

- The switch expression is evaluated and converted to the governing type.
- If one of the constants specified in a case label is equal to the value of the switch expression, control is transferred to the statement list following the matched case label.
- If no constant matches the value of the switch expression and if a default label is present, control is transferred to the statement list following the default label.
- If no constant matches the value of the switch expression and if no default label is present, control is transferred to the end point of the switch statement.

  If the end point of the statement list of a switch section is reachable, a compile-time error occurs. This is known as the "no fall through" rule. Consider the following code.

```
switch (i)
{
   case 0:
            CaseZero();
   break;
case 1:
   CaseOne();
```

```
        break;
    default:
        CaseOthers();
        break;
    }
```

It is a valid code because no switch section has a reachable end point. Unlike C and C++, execution of a switch section is not permitted to "fall through" to the next switch section, therefore, the following code will show error.

```
switch (i)
{
case 0:
  CaseZero();
case 1:
  CaseZeroOrOne();
default:
  CaseAny();
}
```

When execution of a switch section is to be followed by execution of another switch section, an explicit goto case or goto default statement must be used, as shown in the following code.

```
switch (i)
{
case 0:
  CaseZero();
  goto case 1;
case 1:
  CaseZeroOrOne();
  goto default;
default:
  CaseAny();
  break;
}
```

Moreover, multiple labels can be used in a switch-section as shown in the following code.

```
switch (i)
{
case 0:
  CaseZero();
  break;
case 1:
  CaseOne();
  break;
case 2:
default:
  CaseTwo();
  break;
}
```

This code is perfectly legal. It does not violate the "no fall through" rule because the labels case 2: and default: are parts of the same switch-section.

The "no fall through" rule prevents a common class of bugs that occur in C and C++ when break statements are accidentally omitted. Also, because of this rule, the switch sections of a switch statement can be arbitrarily rearranged without affecting the behavior of the statement. For example, the sections of the switch statement above can be reversed without affecting the behavior of the statement (see the code ahead).

```
switch (i)
{
default:
  CaseAny();
  break;
case 1:
  CaseZeroOrOne();
  goto default;
case 0:
  CaseZero();
  goto case 1;
}
```

The statement list of a switch section typically ends in a break, goto case, or goto default statement, but any construct that renders the end point of the statement list unreachable is permitted. For example, a while statement controlled by the boolean expression true is known to never reach its end point. Likewise, a throw or return statement always transfer control elsewhere and never reaches its end point. Thus, the following code is valid.

```
switch (i)
{
case 0:
  while (true) F();
case 1:
  throw new ArgumentException();
case 2:
  return;
}
```

The switch_expression is not always required to be a numeric type. It can also be string type. Consider the following code.

```
void Action(string action)
{
  switch (action.ToLower())
{
  case "run":
    DoRun();
    break;
  case "save":
    DoSave();
    break;
  case "quit":
    DoQuit();
    break;
  default:
    InvalidAction(action);
    break;
  }
}
```

Like the string equality operators, the switch statement is case sensitive and will execute a given switch section only if the switch expression string exactly matches a case label constant. As illustrated by the example above, a switch statement can be made case insensitive by converting the switch expression string to lower case and writing all case label constants in lower case.

When the governing type of a switch statement is string, the value null is permitted as a case label constant.

A switch-block may contain declaration statements. The scope of a local variable or constant declared in a switch block extends from the declaration to the end of the switch block.

Within a switch block, the meaning of a name used in an expression context must always be the same.

The statement list of a given switch section is reachable if the switch statement is reachable and at least one of the following is true:

- The switch expression is a non-constant value.
- The switch expression is a constant value that matches a case label in the switch section.
- The switch expression is a constant value that doesn't match any case label, and the switch section contains the default label.
- A switch label of the switch section is referenced by a reachable goto case or goto default statement.

The end point of a switch statement is reachable if at least one of the following is true:

- The switch statement contains a reachable break statement that exits the switch statement.
- The switch statement is reachable, the switch expression is a non-constant value, and no default label is present.
- The switch statement is reachable, the switch expression is a constant value that doesn't match any case label, and no default label is present.

Student Activity 1

1. What do you understand by labeled statement?
2. What is a block? How is it executed?
3. What are the various mechanisms provided in C# to accomplish branching in a program?
4. What is the problem of "Fall Through" associated with switch-case. How it can be over come in C#?

Looping statements

Very often one or a group of statements are required to be executed a specified number of times or until some condition becomes true/false. Such iterative executions are made possible by looping statements. C# offers the many looping constructs like - while statement, do-statement, for-statement and foreach statement.

5.4 THE WHILE STATEMENT

The while statement has two parts - looping condition and body. The looping condition is a Boolean expression that evaluates to either true or false. The body consists of either a single statement or a block of statements, as shown below.

```
while (boolean_expression)
   statement;
```

The statement executes the statement as long as the Boolean_expression is true. Once the Boolean_expression becomes false, the statement reaches its end.

A while statement is executed as follows:

- The boolean-expression is evaluated.
- If the boolean expression yields true, control is transferred to the embedded statement. When and if control reaches the end point of the embedded statement (possibly from execution of a continue statement), control is transferred to the beginning of the while statement.
- If the boolean expression yields false, control is transferred to the end point of the while statement.

Within the embedded statement of a while statement, a break statement may be used to transfer control to the end point of the while statement (thus ending iteration of the embedded statement),

and a continue statement may be used to transfer control to the end point of the embedded statement (thus performing another iteration of the while statement).

The embedded statement of a while statement is reachable if the while statement is reachable and the boolean expression does not have the constant value false.

The end point of a while statement is reachable if at least one of the following is true:

- The while statement contains a reachable break statement that exits the while statement.
- The while statement is reachable and the boolean expression does not have the constant value true.

Consider the following function that prints numbers from 1 to the input argument.

```
void PrintSeries(int x)
{
  int i = 1;
  while(i <= x)
  {
    System.Console.WriteLine("{0}",i);
    i++;
  }
}
```

Note that if the Boolean_expression happens to be false at the time of executing while statement, the statement will not be executed even once. Consider the code listed below.

```
Int i = 10;
While(i < 5)
  System.Console.WriteLine("Not executed even once");
```

In this case the WriteLine statement will not be executed even once because the condition is false at the time of the entry of while statement.

For the while loop to terminate the loop-controlling condition must be modified in the body. Here is an example of infinite loop.

```
while(true);
```

5.5 THE DO STATEMENT

The do statement executes an embedded statement at least once depending on the given Boolean_expression. The syntax is given below.

```
do
statement
while(boolean-expression);
```

The statement is executed before the Boolean_expression is evaluated. Therefore, the statement is executed at least once. A do statement is executed as follows:

- Control is transferred to the embedded statement.
- When and if control reaches the end point of the embedded statement (possibly from execution of a continue statement), the boolean-expression is evaluated. If the boolean expression yields true, control is transferred to the beginning of the do statement. Otherwise, control is transferred to the end point of the do statement.

Within the embedded statement of a do statement, a break statement may be used to transfer control to the end point of the do statement (thus ending iteration of the embedded statement), and a continue statement may be used to transfer control to the end point of the embedded statement (thus performing another iteration of the do statement).

The embedded statement of a do statement is reachable if the do statement is reachable.

The end point of a do statement is reachable if at least one of the following is true:

- The do statement contains a reachable break statement that exits the do statement.
- The end point of the embedded statement is reachable and the boolean expression does not have the constant value true.

5.6 THE FOR STATEMENT

The for statement evaluates a sequence of initialization expressions and then, while a condition is true, repeatedly executes an embedded statement and evaluates a sequence of iteration expressions. The syntax is given below.

```
for(initializer; condition; iterator)
statement;
```

The for loop has four parts. However all the four parts are optional and they may be missing in a an implementation. The initializer is a comma separated list of initial values. The condition is the looping condition. The for loop executes as long as the looping-condition remains true. The scope of a local variable declared by a for-initializer starts at the variable-declarator for the variable and extends to the end of the embedded statement. The scope includes the for-condition and the for-iterator.

- The for-condition, if present, must be a boolean-expression.
- The for-iterator, if present, consists of a list of statement-expressions separated by commas.

A for statement is executed as follows:

- If a for-initializer is present, the variable initializers or statement expressions are executed in the order they are written. This step is only performed once.
- If a for-condition is present, it is evaluated.
- If the for-condition is not present or if the evaluation yields true, control is transferred to the embedded statement. When and if control reaches the end point of the embedded statement (possibly from execution of a continue statement), the expressions of the for-iterator, if any, are evaluated in sequence, and then another iteration is performed, starting with evaluation of the for-condition in the step above.
- If the for-condition is present and the evaluation yields false, control is transferred to the end point of the for statement.

Within the embedded statement of a for statement, a break statement may be used to transfer control to the end point of the for statement (thus ending iteration of the embedded statement), and a continue statement may be used to transfer control to the end point of the embedded statement (thus executing another iteration of the for statement).

The embedded statement of a for statement is reachable if one of the following is true:

- The for statement is reachable and no for-condition is present.
- The for statement is reachable and a for-condition is present and does not have the constant value false.

The end point of a for statement is reachable if at least one of the following is true:

- The for statement contains a reachable break statement that exits the for statement.
- The for statement is reachable and a for-condition is present and does not have the constant value true.

The following code prints the series of integers from 1 to 100.

```
for(int= i; i <= 100; i++)
  System.Console.WriteLine("{0},i);
```

5.7 THE FOREACH STATEMENT

The foreach statement enumerates the elements of a collection, executing an embedded statement for each element of the collection. Here is the syntax.

```
foreach identifier statement;
```

The identifier of a foreach statement declare the iteration variable of the statement. The iteration variable corresponds to a read-only local variable with a scope that extends over the embedded statement. During execution of a foreach statement, the iteration variable represents the collection element for which iteration is currently being performed. A compile-time error occurs if the embedded statement attempts to assign to the iteration variable or pass the iteration variable as a ref or out parameter.

The type of the expression of a foreach statement must be a collection type (as defined below), and an explicit conversion must exist from the element type of the collection to the type of the iteration variable.

A type C is said to be a collection type if all of the following are true:

- C contains a public instance method with the signature GetEnumerator() that returns a struct-type, class-type, or interface-type, in the following called E.
- E contains a public instance method with the signature MoveNext() and the return type bool.
- E contains a public instance property named Current that permits reading. The type of this property is said to be the element type of the collection type.

The System.Array type is a collection type, and since all array types derive from System.Array, any array type expression is permitted in a foreach statement. For single-dimensional arrays, the foreach statement enumerates the array elements in increasing index order, starting with index 0 and ending with index Length - 1. For multi-dimensional arrays, the indices of the rightmost dimension are increased first.

A foreach statement is executed as follows:

- The collection expression is evaluated to produce an instance of the collection type. This instance is referred to as c in the following. If c is of a reference-type and has the value null, a NullReferenceException is thrown.
- An enumerator instance is obtained by evaluating the method invocation c.GetEnumerator(). The returned enumerator is stored in a temporary local variable, in the following referred to as e. It is not possible for the embedded statement to access this temporary variable. If e is of a reference-type and has the value null, a NullReferenceException is thrown.
- The enumerator is advanced to the next element by evaluating the method invocation e.MoveNext().

If the value returned by e.MoveNext() is true, the following steps are performed:

- The current enumerator value is obtained by evaluating the property access e.Current, and the value is converted to the type of the iteration variable by an explicit conversion. The resulting value is stored in the iteration variable such that it can be accessed in the embedded statement.
- Control is transferred to the embedded statement. When and if control reaches the end point of the embedded statement (possibly from execution of a continue statement), another foreach iteration is performed, starting with the step above that advances the enumerator.

If the value returned by e.MoveNext() is false, control is transferred to the end point of the foreach statement.

Within the embedded statement of a foreach statement, a break statement may be used to transfer control to the end point of the foreach statement (thus ending iteration of the embedded statement), and a continue statement may be used to transfer control to the end point of the embedded statement (thus executing another iteration of the foreach statement).

The embedded statement of a foreach statement is reachable if the foreach statement is reachable. Likewise, the end point of a foreach statement is reachable if the foreach statement is reachable.

Student Activity 2

1. What are the various loops available in C#? Explain with the help of suitable codes.
2. Differentiate with suitable example between do…while and while statements.

5.8 JUMP STATEMENTS

Jump statements unconditionally transfer control to another location. The location to which a jump statement transfers control is called the target of the jump statement. There are five jump statements provided by C#.

```
break-statement
continue-statement
goto-statement
return-statement
throw-statement
```

When a jump statement occurs within a block, and when the target of the jump statement is outside that block, the jump statement is said to exit the block. While a jump statement may transfer control out of a block, it can never transfer control into a block.

Execution of jump statements is complicated by the presence of intervening try statements. In the absence of such try statements, a jump statement unconditionally transfers control from the jump statement to its target. In the presence of such intervening try statements, execution is more complex. If the jump statement exits one or more try blocks with associated finally blocks, control is initially transferred to the finally block of the innermost try statement. When and if control reaches the end point of a finally block, control is transferred to the finally block of the next enclosing try statement. This process is repeated until the finally blocks of all intervening try statements have been executed.

Consider the following code.

```
static void FunctionOne()
{
  while (true)
  {
    try
    {
      try
      {
        Console.WriteLine("Before break");
        break;
      }
      finally
      {
        Console.WriteLine("Innermost finally block");
      }
  }
  finally
  {
    Console.WriteLine("Outermost finally block");
  }
}
  Console.WriteLine("After break");
}
```

Here, the finally blocks associated with two try statements are executed before control is transferred to the target of the jump statement.

5.9 THE BREAK STATEMENT

The break statement exits the nearest enclosing switch, while, do, for, or foreach statement. It syntax consists of nothing but the word break.

```
break;
```

The target of a break statement is the end point of the nearest enclosing switch, while, do, for, or foreach statement. If a break statement is not enclosed by a switch, while, do, for, or foreach statement, a compile-time error occurs.

When multiple switch, while, do, for, or foreach statements are nested within each other, a break statement applies only to the innermost statement. To transfer control across multiple nesting levels, a goto statement must be used.

A break statement cannot exit a finally block. When a break statement occurs within a finally block, the target of the break statement must be within the same finally block, or otherwise a compile-time error occurs.

A break statement is executed as follows:

- If the break statement exits one or more try blocks with associated finally blocks, control is initially transferred to the finally block of the innermost try statement. When and if control reaches the end point of a finally block, control is transferred to the finally block of the next enclosing try statement. This process is repeated until the finally blocks of all intervening try statements have been executed.
- Control is transferred to the target of the break statement.

Because a break statement unconditionally transfers control elsewhere, the end point of a break statement is never reachable.

The continue statement

The continue statement starts a new iteration of the nearest enclosing while, do, for, or foreach statement. The syntax consists of nothing but the word continue.

```
continue;
```

The target of a continue statement is the end point of the embedded statement of the nearest enclosing while, do, for, or foreach statement. If a continue statement is not enclosed by a while, do, for, or foreach statement, a compile-time error occurs.

When multiple while, do, for, or foreach statements are nested within each other, a continue statement applies only to the innermost statement. To transfer control across multiple nesting levels, a goto statement must be used.

A continue statement cannot exit a finally block. When a continue statement occurs within a finally block, the target of the continue statement must be within the same finally block, or otherwise a compile-time error occurs.

A continue statement is executed as follows:

- If the continue statement exits one or more try blocks with associated finally blocks, control is initially transferred to the finally block of the innermost try statement. When and if control reaches the end point of a finally block, control is transferred to the finally block of the next enclosing try statement. This process is repeated until the finally blocks of all intervening try statements have been executed.
- Control is transferred to the target of the continue statement.
- Because a continue statement unconditionally transfers control elsewhere, the end point of a continue statement is never reachable.

5.10 THE GOTO STATEMENT

The goto statement is another branching statement that transfers control of execution to a specified labeled statement. The general syntax of - goto - statement is given below.

```
goto identifier;
Or
goto case constant-expression;
Or
goto default;
```

The target of a goto identifier statement is the labeled statement with the given label. If a label with the given name does not exist in the current function member, or if the goto statement is not within the scope of the label, a compile-time error occurs. This rule permits the use of a goto statement to transfer control out of a nested scope, but not into a nested scope. Consider the following code.

```
class TestClass
{
  static void Main(string[] args)
{
    int i = 0;
    while (true)
{
    Console.WriteLine(i++);
    if (i == 10)
      goto done;
  }
  done:
    Console.WriteLine("Done");
}
}
```

Here, a goto statement is used to transfer control out of a nested scope. The target of a goto case statement is the statement list of the switch section in the nearest enclosing switch statement that contains a case label with the given constant value. If the goto case statement is not enclosed by a switch statement, if the constant-expression is not implicitly convertible to the governing type of the nearest enclosing switch statement, or if the nearest enclosing switch statement does not contain a case label with the given constant value, a compile-time error occurs.

The target of a goto default statement is the statement list of the switch section in the nearest enclosing switch statement that contains a default label. If the goto default statement is not enclosed by a switch statement, or if the nearest enclosing switch statement does not contain a default label, a compile-time error occurs.

A goto statement cannot exit a finally block. When a goto statement occurs within a finally block, the target of the goto statement must be within the same finally block, or otherwise a compile-time error occurs.

A goto statement is executed as follows:

- If the goto statement exits one or more try blocks with associated finally blocks, control is initially transferred to the finally block of the innermost try statement. When and if control reaches the end point of a finally block, control is transferred to the finally block of the next enclosing try statement. This process is repeated until the finally blocks of all intervening try statements have been executed.
- Control is transferred to the target of the goto statement.

Because a goto statement unconditionally transfers control elsewhere, the end point of a goto statement is never reachable.

The - return - statement

This is an unconditional branching statement. The return statement appears in a method. When a method is called on an object the control of execution passes to the first statement of the called method. The called method continues to execute its statements until either it reaches the end of the method or encounters a - return - statement. In either case the control of execution passes on to the caller method. The general syntax of the - return - statement is given below.

```
return;
Or
return expression;
Or
return (expression);
```

The - return - statement may or may not have a following expression. When it does not have any associated expression the return-type of the method must be - void - type. Certain methods, such as set accessor of a property, indexers, constructors and destructors also do not have any return expression. In all other cases, the expression associated with- return - statement must be of the return-type of the method.

If the computed value of the type of return expression is different from the return-type of the method, the return-expression must be explicitly converted into the return-type by appropriate type-casting.

Obviously, a method that returns a data-type must have at least one reachable - return - statement. And finally, a return statement cannot appear in a - finally - block of a method.

The throw statement

The throw statement throws an exception.

```
throw expression;
```

A throw statement with an expression throws the exception produced by evaluating the expression. The expression must denote a value of the class type System.Exception or of a class type that derives from System.Exception. If evaluation of the expression produces null, a NullReferenceException is thrown instead.

Note that expression in the throw statement is optional. A throw statement with no expression can be used only in a catch block. It re-throws the exception that is currently being handled by the catch block.

Because a throw statement unconditionally transfers control elsewhere, the end point of a throw statement is never reachable.

When an exception is thrown, control is transferred to the first catch clause in a try statement that can handle the exception. The process that takes place from the point of the exception being thrown to the point of transferring control to a suitable exception handler is known as exception propagation. Propagation of an exception consists of repeatedly evaluating the following steps until a catch clause that matches the exception is found. In the descriptions, the throw point is initially the location at which the exception is thrown.

In the current function member, each try statement that encloses the throw point is examined. For each statement S, starting with the innermost try statement and ending with the outermost try statement, the following steps are evaluated:

- If the try block of S encloses the throw point and if S has one or more catch clauses, the catch clauses are examined in order of appearance to locate a suitable handler for the exception. The first catch clause that specifies the exception type or a base type of the exception type is considered a match. A general catch clause is considered a match for any exception type. If a matching catch clause is located, the exception propagation is completed by transferring control to the block of that catch clause.
- Otherwise, if the try block or a catch block of S encloses the throw point and if S has a finally block, control is transferred to the finally block. If the finally block throws another exception,

processing of the current exception is terminated. Otherwise, when control reaches the end point of the finally block, processing of the current exception is continued.

- If an exception handler was not located in the current function member invocation, the function member invocation is terminated. The steps above are then repeated for the caller of the function member with a throw point corresponding to the statement from which the function member was invoked.
- If the exception processing ends up terminating all function member invocations in the current thread or process, indicating that the thread or process has no handler for the exception, then the thread or process is itself terminated in an implementation-defined fashion.

The try statement

The try statement provides a mechanism for catching exceptions that occur during execution of a block. The try statement furthermore provides the ability to specify a block of code that is always executed when control leaves the try statement. The syntax is given below.

```
try   block   catch-clauses
try   block   finally-clause
try   block   catch-clauses   finally-clause
```

There are three possible forms of try statements:

- A try block followed by one or more catch blocks.
- A try block followed by a finally block.
- A try block followed by one or more catch blocks followed by a finally block.

When a catch clause specifies a class-type, the type must be System.Exception or a type that derives from System.Exception.

When a catch clause specifies both a class-type and an identifier, an exception variable of the given name and type is declared. The exception variable corresponds to a local variable with a scope that extends over the catch block. During execution of the catch block, the exception variable represents the exception currently being handled. For purposes of definite assignment checking, the exception variable is considered definitely assigned in its entire scope.

Unless a catch clause includes an exception variable name, it is impossible to access the exception object in the catch block.

A catch clause that specifies neither an exception type nor an exception variable name is called a general catch clause. A try statement can only have one general catch clause, and if one is present it must be the last catch clause.

Though the throw statement is restricted to throwing exceptions of type System.Exception or a type that derives from System.Exception, other languages are not bound by this rule, and so may throw exceptions of other types. A general catch clause can be used to catch such exceptions, and a throw statement with no expression can be used to re-throw them.

An error occurs if a catch clause specifies a type that is the same as or derived from a type that was specified in an earlier catch clause. Because catch clauses are examined in order of appearance to locate a handler for an exception, without this restriction it would be possible to write unreachable catch clauses.

Within a catch block, a throw statement with no expression can be used to re-throw the exception that was caught by the catch block. Assignments to an exception variable do not alter the exception that is re-thrown.

Consider the following code.

```
class TestClass
{
  static void FunctionOne()
{
```

```
      try
{
      FunctionTwo();
   }
   catch (Exception e)
{
      Console.WriteLine("Exception in FunctionOne: " + e.Message);
      e = new Exception("FunctionOne");
      throw;
// re-throw
}
}
static void FunctionTwo()
{
   throw new Exception("FunctionTwo");
}
static void Main()
{
   try
{
      FunctionOne();
   }
   catch (Exception e)
{
      Console.WriteLine("Exception in Main: " + e.Message);
   }
}
}
```

Here, the method FunctionOne catches an exception, writes some diagnostic information to the console, alters the exception variable, and re-throws the exception. The exception that is re-thrown is the original exception, so the output of the program is:

Exception in FunctionOne: FunctionTwo

Exception in Main: FunctionTwo

It is an error for a break, continue, or goto statement to transfer control out of a finally block. When a break, continue, or goto statement occurs in a finally block, the target of the statement must be within the same finally block, or otherwise a compile-time error occurs.

It is an error for a return statement to occur in a finally block.

A try statement is executed as follows:

- Control is transferred to the try block.

When and if control reaches the end point of the try block:

- If the try statement has a finally block, the finally block is executed.
- Control is transferred to the end point of the try statement.

If an exception is propagated to the try statement during execution of the try block:

- The catch clauses, if any, are examined in order of appearance to locate a suitable handler for the exception. The first catch clause that specifies the exception type or a base type of the exception type is considered a match. A general catch clause is considered a match for any exception type. If a matching catch clause is located:
  1. If the matching catch clause declares an exception variable, the exception object is assigned to the exception variable.

2. Control is transferred to the matching catch block.
3. When and if control reaches the end point of the catch block:
4. If the try statement has a finally block, the finally block is executed.
5. Control is transferred to the end point of the try statement.
6. If an exception is propagated to the try statement during execution of the catch block:
7. If the try statement has a finally block, the finally block is executed.
8. The exception is propagated to the next enclosing try statement.

- If the try statement has no catch clauses or if no catch clause matches the exception:
  1. If the try statement has a finally block, the finally block is executed.
  2. The exception is propagated to the next enclosing try statement.

The statements of a finally block are always executed when control leaves a try statement. This is true whether the control transfer occurs as a result of normal execution, as a result of executing a break, continue, goto, or return statement, or as a result of propagating an exception out of the try statement.

If an exception is thrown during execution of a finally block, the exception is propagated to the next enclosing try statement. If another exception was in the process of being propagated, that exception is lost. The process of propagating an exception is further discussed in the description of the throw statement.

- The try block of a try statement is reachable if the try statement is reachable.
- A catch block of a try statement is reachable if the try statement is reachable.
- The finally block of a try statement is reachable if the try statement is reachable.

The end point of a try statement is reachable if both of the following are true:

- The end point of the try block is reachable or the end point of at least one catch block is reachable.
- If a finally block is present, the end point of the finally block is reachable.

5.11 THE CHECKED AND UNCHECKED STATEMENTS

The checked and unchecked statements are used to control the overflow checking context for integral-type arithmetic operations and conversions. The syntax is given below.

checked block

unchecked block

The checked statement causes all expressions in the block to be evaluated in a checked context, and the unchecked statement causes all expressions in the block to be evaluated in an unchecked context.

The checked and unchecked statements are precisely equivalent to the checked and unchecked operators, except that they operate on blocks instead of expressions.

The lock statement

The lock statement obtains the mutual-exclusion lock for a given object, executes a statement, and then releases the lock. The syntax is given below.

```
lock(expression)
statement;
```

The expression of a lock statement must denote a value of a reference-type. An implicit boxing conversion is never performed for the expression of a lock statement, and thus it is an error for the expression to denote a value of a value-type.

A lock statement of the form

lock (x) …

where x is an expression of a reference-type, is precisely equivalent to

```
System.Threading.Monitor.Enter(x);
try
{
  …
}
finally
{
  System.Threading.Monitor.Exit(x);
}
```

except that x is only evaluated once. The exact behavior of the Enter and Exit methods of the System.Threading.Monitor class is implementation-defined.

The System.Type object of a class can conveniently be used as the mutual-exclusion lock for static methods of the class (see the following code).

```
class CacheClass
{
  public static void Add(object x)
{
     lock (typeof(CacheClass))
{
     …
  }
}
public static void Remove(object x)
{
  lock (typeof(CacheClass))
{
     …
  }
}
}
```

The using statement

The using statement obtains one or more resources, executes a statement, and then disposes of the resource. The syntax is given below.

```
using(resource-acquisition) statement;
```

A resource is a class or struct that implements System.IDisposable, which includes a single parameter-less method named Dispose. Code that is using a resource can call Dispose to indicate that the resource is no longer needed. If Dispose is not called, then automatic disposal eventually occurs as a consequence of garbage collection.

If the form of resource-acquisition is local-variable-declaration then the type of the local-variable-declaration must be System.IDisposable or a type that can be implicitly converted to System.IDisposable. If the form of resource-acquisition is expression then this expression must be System.IDisposable or a type that can be implicitly converted to System.IDisposable.

Local variables declared in a resource-acquisition are read-only, and must include an initializer.

A using statement is translated into three parts: acquisition, usage, and disposal. Usage of the resource is enclosed in a try statement that includes a finally clause. This finally clause disposes of the resource. If a null resource is acquired, then no call to Dispose is made, and no exception is thrown.

For example, using statement of the form

```
using (R r1 = new R ())
{
  r1.FunctionOne();
}
is precisely equivalent to
R r1 = new R();
try
{
  r1.FunctionOne();
}
finally
{
  if (r1 != null) ((IDisposable)r1).Dispose();
}
```

Student Activity 3

1. What is the purpose of a jump statement?
2. Explain the try statement. What are the three possible forms of try statement?

5.12 SUMMARY

1. C# provides a rich variety of statements that include all the familiar statements of language C and C++.
2. The switch statement is another convenient tool provided by C# to handle the situations in which multiple decisions to be made on an expression that can have multiple values.
3. Conditional operator could be used as an alternate to the if else statement as both of them provides a control flow capacity.
4. The For loop in C# is the simplest, fixed and entry controlled loop.
5. The second type of loop, the while loop is an entry controlled loop as it tests the conditions first and if the condition us true, then only the control will enter into the loop body.
6. An empty loop can also be configured using while statement and could be used as a time delay loop.
7. Unlike the For and while loops, a do while loop always executes at least once.
8. The Foreach statement enumerates the elements of a collections, executing an embedded statement for each element of the collection.
9. Jump statement unconditionally transfer control without checking any condition.

5.13 KEYWORDS

- **Blocks:** Multiple statements blocked to make them behave as a single statement. A block may contain declaration statements.
- **Control Statement:** The statement by which we can control the flow of the program execution is called as control statement.
- **Switch Statement:** It is a multiple branch statement that successively tests the value of an expression against a list of case values and when a match is found, the statement associated with the particular case is executed.
- **Loop Body:** The loop body consists of statement(s) that is supposed to be executed again and again as long as the condition expression evaluates to true.
- **Jump statement:** These statements unconditionally transfer control to another location referred to as the target of the jump statement. A jump statement may transfer control out of a block but it can never transfer control into a block.

5.14 REVIEW QUESTIONS

1. Compare if-else statement with the switch statement. Explain how the switch statement provides a more elegant way to handle multi-way branching.
2. Translate a For loop into a while loop and vice-versa.
3. Explain with an example how a goto statement is used to transfer control out of a nested scope.

5.14 FURTHER READINGS

CLR via C#, Second Edition (Paperback) by Jeffrey Richter, Publisher: Microsoft Press; 2nd edition (February 22, 2006)

Essential C# 2.0 (Microsoft .NET Development Series) (Paperback) by Mark Michaelis, Publisher: Addison-Wesley Professional (July 13, 2006)

C# Programming Language, The (2nd Edition) (Microsoft .NET Development Series) (Hardcover) by Anders Hejlsberg, Scott Wiltamuth, Peter Golde, Publisher: Addison-Wesley Professional; 2 edition (June 9, 2006)

Programming in the Key of C# (Paperback) by Charles Petzold, Publisher: Microsoft Press; 1 edition (July 30, 2003)

UNIT

6

EXCEPTIONS

L E A R N I N G O B J E C T I V E S

After studying this unit, you should be able to

- Define exception in C# programming
- Understand handling exceptions
- Understand about throw and finally clause
- Define system exception class

U N I T S T R U C T U R E

6.1 INTRODUCTION

This chapter will introduce you to what exceptions are and how they are handled in C3 by using system exceptions that are designed for the purpose of handling errors. Handling of runtime errors is one of the most crucial tasks ahead of any programmer. As a programmer one must make provisions in case exceptions occur for the program to terminate or provide an alternative course of action.

6.2 EXCEPTIONS

There are many ways in which a program under execution may commit error. For instance, a program may attempt to divide a number by zero at some point of time in execution. A runtime error in a program is referred to as an exception. Clearly it is an undesirable trait of a program. Note that exceptions are errors occurring at runtime. The compiler will not detect them.

One can take extra caution to include codes to avoid such conditions that give rise to exceptions. Howsoever one may be careful, exceptions cannot be ruled out.

Therefore, the programmer must make provisions for the program to terminate gracefully or to take alternative course of action in case an exception does occur. Traditionally, as in C programming, if-else-if ladder used to be employed for this purpose. This approach often renders the program utterly unreadable. Object-oriented programming frameworks including .NET have introduced an elegant way of catching and handling exceptions.

If an exception has its origin in an application program it is called application level exception. On the other hand if it is originated in the system program on the top of which the application program in question runs it is called system level exception.

Here we shall be dealing with the way C# handles exceptions though the approach is quite general, idiosyncrasy apart.

C# provides a well-defined, structured, uniform, and type-safe way of handling exceptions at both levels - application and system. An exception is represented by a class in C#. Exception class comes under the system namespace. C# uses try-catch-throw mechanism to handle exceptions. Simply stated it reads as: try (run under supervision) a block of code; in case of an exception catching it; create an object of it; and then throw it for some other object to handle it. Thus, an exception must be represented by an instance of System.Exception or any other class derived from System.Exception class. This is in contrast to languages like C++ where this restriction is not applicable and therefore any value of any type can be used to represent an exception.

Another notable improvement is regarding the finally block. A finally block always executes irrespective of the condition normal or exceptional in C# programs. System-level exceptions are defined by different classes. Thus we have exception classes corresponding to overflow, divide-by-zero, and null dereferences exceptions.

6.3 HANDLING EXCEPTIONS

A C# program may throw an exception. The thrown exception may be handled by the program itself, however, it is not necessary. If the program that throws an exception also includes code to handle it, the exception is known as checked-exception. On the contrary, if the program simply throws an exception and allows the runtime environment to take the necessary actions in response, it is called unchecked exception.

For instance, division by zero is an unchecked exception. It is the runtime environment that handles this exception implying that a programmer may rely on the runtime system for its handling. Similarly, when a program code references an array with an index for which no array element exists, it causes an unchecked exception.

Consider a numeric variable in a program whose value is being read from the keyboard. If the value entered is non-numeric. This will indicate an exception. This exception is an example of checked exception because of course you would not like to leave this to the runtime environment. Instead you would like to insert code in the program itself to take the necessary action such as prompting the user to re-enter a valid numerical value.

Consider the following code snippet.

```
class ExceptionExample1
{
int x = 100;

public static void Main()
{
System.Console.WriteLine("Enter the divisor:");
int y = System.Convert.ToInt32(System.Console.ReadLine());
double r = x/y;
System.Console.WriteLine("Result = "+r);
}
}
```

Look at the output for a couple of runs.

Enter the divisor:

You entered 20. The output would be:

```
   Result = 5
Run it again.
```

Enter the divisor:

Enter 0 this time. Look at the output as shown below.

An exception 'System.DivideByZeroException' has occurred in ExceptionExample1.exe.

Note how the in-built exception System.DivideByZeroException was thrown by the program and subsequently caught by the runtime environment. In case you decide that your program catches this exception you will have to insert necessary codes into your program.

6.4 TRY AND CATCH CLAUSES

In a C# program, try defines a code block which is likely to throw an exception. It is the onus on the programmer to anticipate which statements are capable of throwing an exception and which the program needs to catch. The codes included in the try clause may or may not throw an exception.

A try clause is associated with a catch block. This block catches the exception thrown by the try block and performs the actions specified in there.

Let us apply this mechanism to handle the exception thrown in the above program snippet. We want to dispatch a meaningful error message if division by zero is attempted. Enclose the statements in a try block (see the listing below).

```
class ExceptionExample2
{
int x = 100;

public static void Main()
{
try
{
System.Console.WriteLine("Enter the divisor:");
int y = System.Convert.ToInt32(System.Console.ReadLine());
double r = x/y;
System.Console.WriteLine("Result = "+r);
}
catch (System.DivideByZeroException exp)
{
  System.Console.WriteLine("Error: Division by zero not allowed!");
  System.Console.WriteLine("Action: Program will terminate!);
}
}
```

Now this program catches the exception thrown by the try code (in this case System System.DivideByZeroException). When this exception is thrown by the try block, an object of the System.DivideByZeroException exception class is created (exp in this case) which is caught by the associated catch block and the program terminates. Our catch clause simply writes two lines of message to the console in response before terminating. This is certainly a better way of handling the exception than the previous one. To see the difference, run the program and input 0 as divisor. The following will be the output.

```
Enter the divisor:0
```

You enter 0. The program gives you the following output.

```
Error: Division by zero not allowed!
Action: Program will terminate!
```

Well! Now your program is equipped to catch the System.DivideByZeroException exception. What about other exceptions that this try block may throw. The answer is simple. You can associate more than one catch block to a single try block. For example, our program expects that you enter an integer for the input. What happens when you enter a fraction number instead of

integer? To see the effect run the program ExceptionExample2.exe and input 1.5. The output will be as shown below.

```
Enter the divisor:1.5
```

An exception 'System.FormatException' has occurred in ExceptionExample2.exe.

This exception (System.FormatException) was not handled by the program therefore the runtime has issued this error message. If you want your program to trap this error as well, attach one more catch clause for this exception (see the listing for ExceptionExample3 given below).

```
class ExceptionExample3
{
int x = 100;

public static void Main()
{
try
{
System.Console.WriteLine("Enter the divisor:");
int y = System.Convert.ToInt32(System.Console.ReadLine());
double r = x/y;
System.Console.WriteLine("Result = "+r);
}
catch (System.DivideByZeroException exp)
{
  System.Console.WriteLine("Error: Division by zero not allowed!");
  System.Console.WriteLine("Action: Program will terminate!);
}
catch (System.FormatException exp)
{
  System.Console.WriteLine("Error: Non-integer input not allowed!");
  System.Console.WriteLine("Action: Program will terminate!);
}
}
```

Now run the program ExceptionExample3.exe with input 1.5. The output will be as shown below.

```
Enter the divisor:1.5

Error: Non-integer input not allowed!
Action: Program will terminate!
```

Here you have also captured FormatException in addition to DivideByZero exception. This way you can make your program handle other exceptions. If you do not know which other exceptions may be thrown, you can include specify all the other exceptions by the class - System.Exception. The modified program is listed as ExceptionExample4 below.

```
class ExceptionExample4
{
int x = 100;

public static void Main()
{
try
{
System.Console.WriteLine("Enter the divisor:");
int y = System.Convert.ToInt32(System.Console.ReadLine());
double r = x/y;
System.Console.WriteLine("Result = "+r);
}
catch (System.DivideByZeroException exp)
```

```
{
  System.Console.WriteLine("Error: Division by zero not allowed!");
  System.Console.WriteLine("Action: Program will terminate!);
}
catch (System.FormatException exp)
{
  System.Console.WriteLine("Error: Non-integer input not allowed!");
  System.Console.WriteLine("Action: Program will terminate!);
}
catch (System.Exception exp)
{
  System.Console.WriteLine("Error: Some error has occured!");
  System.Console.WriteLine("Action: Program will terminate!);
}
}
```

Note that the name of the exception object being thrown in all the cases has been kept as exp. This however is not necessary. You can very well pick the names exp1, exp2, exp3 etc. Which exp represents which object depends on the catch block that traps it.

Student Activity 1

1. What is an Exception? Write a program to explain the concept of Exceptional Handling.
2. Explain checked and unchecked exception.
3. Can a try block followed by multiple catch statement? Justify your answer.

6.5 THROW CLAUSE

A C# program throws an exception whenever an abnormal condition arises in the program. However, the programmer is also at the liberty of throwing an exception if he desires so. This is accomplished by throw clause.

Throw clause when executed throws the specified exception unconditionally whether or not any exception has occurred. Look at the following program (ExceptionExample5) in which the programmer throws an exception when th input does not match with the string stored in a variable.

```
class ExceptionExample5
{
string name;
public static void Main()
{
try
{
System.Console.WriteLine("Which is the highest mountain peak in the
world?");
name = System.Console.ReadLine();
if (name != "Everest")
throw new System.Exception();
else
System.Console.WriteLine("Right answer!");
}
catch(Exception e)
  {
  System.Console.WriteLine(e.ToString()+"Sorry! It is the Everst");
  }
  }
}
```

Run the program ExceptionExample5.exe. You will see the following.

Which is the highest mountain peak in the world?Kanchenjunga

The output will be:

Sorry! It is the Everest

You must have realized by now that throw serves more purposes than it meets the eyes.

6.6 FINALLY CLAUSE

At times one wants to execute a piece of code irrespective of whether an exception occurred or not. Such codes are included in the finally clause. Note that the catch clause code is executed only when an appropriate exception occurs but finally clause codes are always executed. See the following program ExceptionExample6 listed below.

```
class ExceptionExample6
{
string name;
public static void Main()
{
try
{
System.Console.WriteLine("Which is the highest mountain peak in the
world?");
name = System.Console.ReadLine();
if (name != "Everest")
throw new System.Exception();
else
System.Console.WriteLine("Right answer!");
}
catch(Exception e)
  {
  System.Console.WriteLine(e.ToString()+"Sorry! It is the Everst");
  }
finally
{
System.Console.WriteLine("It is Nepal ");
    }
  }
}
```

Run the program ExceptionExample6.exe with correct answer. The output will be:

Which is the highest mountain peak in the world?Everest

Right answer!

It is in Nepal

Run the program ExceptionExample6.exe again this time with incorrect answer. The output will be:

Which is the highest mountain peak in the world?Kanchenjunga

Sorry! It is Everest

It is in Nepal

Observe that in both the cases, whether the exception was thrown or not, the finally clause has executed.

Student Activity 2

1. Explain the Throw clause with the help of suitable example.
2. What is the use of finally statement? Explain with the help of suitable example.

6.7 USER DEFINED EXCEPTION

Though in-built exceptions serve the purpose very well, there are situations where the programmer needs to create his own exceptions. All the user-defined exceptions class must derive from the System.Exception base class or any other derivable class which itself derives from System.Exception class. In the following example - ExceptionExample7 - this concept is elucidated. Here we will create an exception class xceptionExample7 having two constructors - one without any parameter and the other having a parameter name.

```
public class MyException : System.Exception
{
public
MyException()
{
System.Console.WriteLine("The integer entered is more than 10");
}
}

class ExceptionExample7
{
  public static void Main()
  {
    int x;
System.Console.WriteLine("Enter an integer");
x = System.Convert.ToInt32(System.Console.ReadLine());
try
{
if(x > 10) throw new MyException();
    }
    catch(MyException e)
    {
    }
  }
}
```

When you run the program ExceptionExample7.exe and input integer 12, an exception of MyException class is thrown which is caught by the catch block and consequently you will see the following output on the console.

```
The integer entered is more than 10
```

Note that the catch clause does not print any message. It is the exception object that issues the message. If you want you can also add some code into the catch clause itself. Also note that by catching System.Exception class will catch all the exceptions occurring in the corresponding try block because all the exceptions are direct or indirect derivative of System.Exception class.

An exception may be thrown multiple times. Thus the catch block can again throw the caught exception as shown in the following program listing ExceptionExample8. Here the caught exception is passed to the base class of the exception class in which the current exception was caught.

```
class ExceptionExample8
{
  public static void Main()
  {
    int x;
```

```
System.Console.WriteLine("Enter an integer");
x = System.Convert.ToInt32(System.Console.ReadLine());
try
{
if(x > 10) throw new MyException();
     }
     catch(MyException e)
     {
       throw e;
     }
   }
}
```

6.8 SYSTEM.EXCEPTION CLASS

As you have seen every exception must derive from the base class System.Exception class. Here is a synopsis of some of the important members and methods.

- Message is a read-only property that contains a human-readable description of the reason for the exception.
- InnerException is a read-only property that contains the "inner exception" for this exception. If this is not null, this indicates that the current exception was thrown in response to another exception. The exception that caused the current exception is available in the InnerException property.
- The value of these properties can be specified in the constructor for System.Exception.

The public properties and methods of System.Exception are listed below.

```
bool System.Exception.Equals(System.Object)
```

This method is inherited from System.Object class which incidentally is the root of all the classes in C#. The method compares the object passed as parameter to the current System.Exception object

```
System.Exception GetBaseException()
```

When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.

```
int System.Exception.GetHashCode()
```

This methods returns to Serves as a hash function for a particular type. It is suitable for use in hash algorithms and data structures like hash table.

```
void GetObjectdata(System.Runtime.SerializationInfo info,
System.Runtime.Serialization.StreamingContext context)
```

When overridden in a derived class, sets the System.Runtime.SerializationInfo with information about the exception.

```
string HelpLink
```

Gets or sets the link to the help file associated with this exception.

```
System.Exception InnerException
```

Gets the System.Exception instance that caused the current exception.

```
string Message
```

Gets a message that describes the current exception.

```
string Source
```

Gets or sets the name of the application or object that caused the exception.

```
string Stacktrace
```

Gets a string representation of the frames on the call stack at the time the current exception was thrown.

```
System.Reflection.MethodBase TargetSite
```

Gets the method that throws the current exception.

Common Exception Classes

Though the list of in-built exceptions in C# is very large, here is a synopsis of some of the important ones.

```
System.OutOfMemoryException
```

Thrown when an attempt to allocate memory (via new) fails.

```
System.StackOverflowException
```

Thrown when the execution stack is exhausted by having too many pending method calls; typically indicative of very deep or unbounded recursion.

```
System.NullReferenceException
```

Thrown when a null reference is used in a way that causes the referenced object to be required.

```
System.TypeInitializationException
```

Thrown when a static constructor throws an exception, and no catch clauses exists to catch in.

```
System.InvalidCastException
```

Thrown when an explicit conversion from a base type or interface to a derived types fails at run time.

```
System.ArrayTypeMismatchException
```

Thrown when a store into an array fails because the actual type of the stored element is incompatible with the actual type of the array.

```
System.IndexOutOfRangeException
```

Thrown when an attempt to index an array via an index that is less than zero or outside the bounds of the array.

```
System.MulticastNotSupportedException
```

Thrown when an attempt to combine two non-null delegates fails, because the delegate type does not have a void return type.

```
System.ArithmeticException
```

A base class for exceptions that occur during arithmetic operations, such as DivideByZeroException and OverflowException.

```
System.DivideByZeroException
```

Thrown when an attempt to divide an integral value by zero occurs.

```
System.OverflowException
```

Thrown when an arithmetic operation in a checked context overflows.

Student Activity 3

1. Explain the concept of user-defined exception with the help of suitable example.
2. List down some of important in-built exception classes.

6.9 SUMMARY

- Trapping and handling of runtime errors is one of the most crucial tasks ahead of any programmer. Programmers must make provisions for the program to terminate gracefully or take alternative course of action in case an exception does occur.
- C# provides three keywords try, catch and finally to do exception handling.
- The finally can be used for doing any clean up process.

- If any exception occurs inside the try block, the control transfers to the appropriate catch block and later to the finally block.
- If there is no exception occurred inside the try block, the control directly transfers to finally block.
- The user-defined exception classes must inherit from either Exception class or one of its standard derived classes.
- If code catches an exception that it isn't going to handle, consider whether it should wrap that exception with additional information before re-throwing it.

6.10 KEYWORDS

- **Exception:** An exception is an error condition or unexpected behavior encountered by an executing program during runtime.
- **Exception Handling:** It is an in-built mechanism in .NET framework to detect and handle run time errors.
- **Try and catch statement:** The try-catch statement consists of a try block followed by one or more catch clauses, which specify handlers for different exceptions.
- **Throw statement:** The throw statement is used to signal the occurrence of an anomalous situation (exception) during the program execution.
- **Finally clause:** The finally block is useful for cleaning up any resources allocated in the try block as well as running any code that must execute even if there is an exception. Control is always passed to the finally block regardless of how the try block exits

6.11 REVIEW QUESTIONS

1. How is an 'Exception' different from 'Error'?
2. Describe the members of 'Exception Class'.
3. 'Exception' have multiple 'Catch Statements'. If yes explain.
4. Write a program to explain the concept of Exceptional Handling?
5. Write a program to accept 10 numbers in an array and print the sorted list via incorporating the concept of exception handling.

6.12 FURTHER READINGS

CLR via C#, Second Edition (Paperback) by Jeffrey Richter, Publisher: Microsoft Press; 2nd edition (February 22, 2006)

C# Programming Language, The (2nd Edition) (Microsoft .NET Development Series) (Hardcover) by Anders Hejlsberg, Scott Wiltamuth, Peter Golde, Publisher: Addison-Wesley Professional; 2 edition (June 9, 2006)

UNIT

7

INHERITANCE AND POLYMORPHISM

L E A R N I N G O B J E C T I V E S

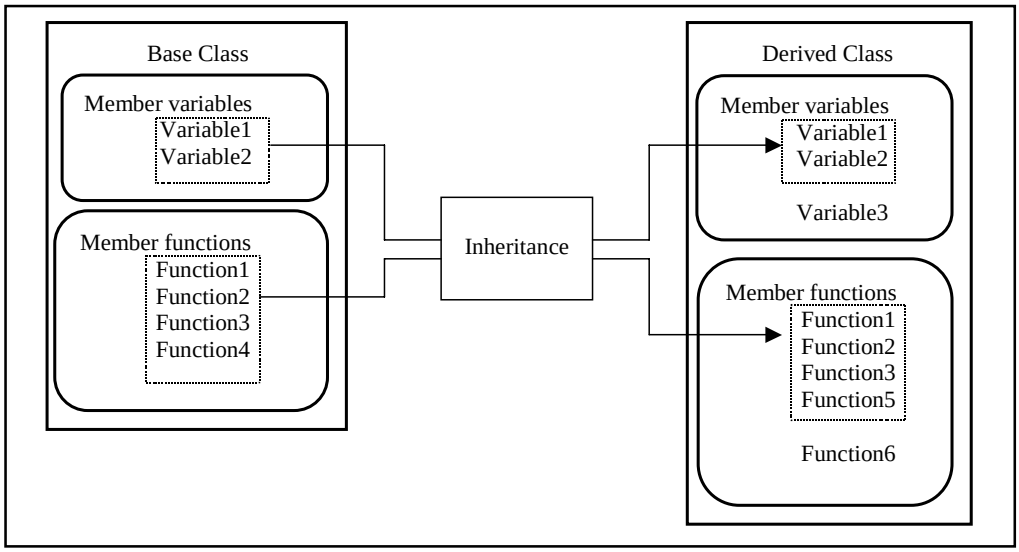
After studying this unit, you should be able to

- Define inheritance
- Define polymorphism
- Correlate base class and derived class
- Understand about operator overloading

U N I T S T R U C T U R E

7.1 INTRODUCTION

Object oriented programming allows reusability of the codes written in a class. This is achieved through inheritance. A new class can be constructed using an existing class in such a way that the new class retains some or all the members of the existing class and may add one or more members to it. This process through which the members of an existing class are obtained by a new class is called inheritance much in the same way as a son inherits his father's property. The new class being created is called the derived class and the existing class whose members are used in the new class is called the base class. The relationship between the base and derived classes is depicted in the following figure.



7.2 BASE CLASS AND DERIVED CLASS

The figure illustrates that a derived class can inherit features (member variables and member functions) of the base and can have its own features (member variables and member functions).

Inheritance follows certain rules that determine which features are inherited and which are not. These rules are summarized below.

- The derived class does not inherit private members of the base class.
- Inheritance is transitive. If C is derived from B, and B is derived from A, then C inherits the members declared in B as well as the members declared in A.
- A derived class extends its direct base class. A derived class can add new members to those it inherits, but it cannot remove the definition of an inherited member.
- Constructors and destructors are not inherited, but all other members are, regardless of their declared accessibility. However, depending on their declared accessibility, inherited members may not be accessible in a derived class.
- A derived class can hide inherited members by declaring new members with the same name or signature. Note however that hiding an inherited member does not remove the member-it merely makes the member inaccessible in the derived class.
- An instance of a class contains a copy of all instance fields declared in the class and its base classes, and an implicit conversion exists from a derived class type to any of its base class types. Thus, a reference to a derived class instance can be treated as a reference to a base class instance.
- A class can declare virtual methods, properties, and indexers, and derived classes can override the implementation of these function members. This enables classes to exhibit polymorphic behavior wherein the actions performed by a function member invocation varies depending on the run-time type of the instance through which the function member is invoked.
- Classes support single inheritance, and the type object is the ultimate base class for all classes.

Inheritance is one of the key concepts of Object Oriented Programming. By using the concept of inheritance, it is possible to create a new class from an existing one and add new features to it. Thus inheritance provides a mechanism for class level re usability. Obviously C# supports inheritance. The syntax of inheritance is very simple and straightforward.

```
class BaseClass
{
}
class DerivedClass : BaseClass
{
}
```

The classes shown in earlier examples all implicitly derive from object. Consider the following code.

```
class One
{
  public void FunctionOne()
{

Console.WriteLine("One.Function");
}
}
```

It shows a class One that implicitly derives from object. Consider the following code.

```
class Two : One
{
  public void FunctionTwo()
{
Console.WriteLine("Two.FunctionTwo");
}
}
class TestClass
{
  static void Main()
{
    Two b = new Two();
    b.FunctionOne();       // Inherited from One
    b.FunctionTwo();       // Introduced in Two

    One a = b;         // Treat a Two as an One
    a.FunctionOne();
  }
}
```

The code shows a class Two that derives from One. The class Two inherits One's FunctionOne method, and introduces a FunctionTwo method of its own.

The operator ':'is used to indicate that a class is inherited from another class. Remember that in C#, a derived class can't be more accessible than it's base class. That means that it is not possible to declare a derived class as public, if it inherits from a private class. For example the following code will generate a compile time error.

```
class BaseClass
{
}

public class DerivedClass : BaseClass
{
}
```

In the above case the BaseClass class is private. We try to inherit a public class from a private class.

In the code listed below DerivedClass inherits public members of the BaseClass x,y and Method(). The objects of the DerivedClass can access these inherited members along with its own member z.

```
using System;
class BaseClass
{
  public int x = 100;
  public int y = 200;
  public void Method()
  {
    Console.WriteLine("BaseClass Method");
  }
}
class DerivedClass : BaseClass
{
  public int z = 300;
}
class TestClass
{
```

```
  public static void Main()
  {
    DerivedClass d1 = new DerivedClass();
    Console.WriteLine("{0},{1},{2}",d1.x,d1.y,d1.z);
    // displays 10,20,30
    d1.Method();
  // displays 'BaseClass Method'
  }
}
```

Methods, properties, and indexers can be virtual, which means that their implementation can be overridden in derived classes (see the following code).

```
using System;
class One
{
  public virtual void FunctionOne()
{
Console.WriteLine("One.FunctionOne");
}
}
class Two: One
{
  public override void FunctionOne()
{
    base.FunctionOne();
    Console.WriteLine("Two.FunctionOne");
  }
}
class TestClass
{
  static void Main()
{
    Two b = new Two();
    b.FunctionOne();
    One a = b;
    a.FunctionOne();
  }
}
```

The code shows a class One with a virtual method FunctionOne, and a class Two that overrides FunctionOne. The overriding method in Two contains a call base.FunctionOne() which calls the overridden method in One.

The abstract modifier allows a class to indicate that it is incomplete and that is intended only as a base class. A class using abstract modifier is called an abstract class. An abstract class can specify abstract members-members that a non-abstract derived class must implement. Consider the following code.

```
using System;
abstract class One
{
  public abstract FunctionOne();
}
class Two: One
{
  public override FunctionOne()
{
```

```
Console.WriteLine("Two.FunctionOne");
}
}
class TestClass
{
  static void Main()
{
     Two b = new Two();
     Two.FunctionOne();
     One a = b;
     a.FunctionOne();
  }
}
```

The code introduces an abstract method FunctionOne in the abstract class One. The non-abstract class Two provides an implementation for this method.

Student Activity 1

1. What do you mean by Inheritance? What are the rules that determine which features are inherited and which are not?
2. What are the benefits of implementing inheritance in object programming?

Hiding through inheritance

It must be noted carefully that inheritance may lead to name hiding. It happens when classes or structs redeclare names that were inherited from base classes. Following rules govern name hiding during inheritance.

- If the name of a constant, field, property, event, or type introduced in a class or struct is the same as that in the base class, the base class member's name becomes hidden.
- A method in a class having same name as a method in the base class, hides the method of the base class provided their signatures are same.
- An indexer in a class or struct hides all base class indexers with the same signature.
- Name hiding does not apply on operators. Thus, an operator in a derived class never hides the operator in the base class.

A warning is sounded by the compiler if the name hiding takes place in a class as shown in the code below.

```
class One
{
  public void FunctionOne()
{}
}
class Two: One
{
  public void FunctionOne()
{}       // Warning, hiding an inherited name
}
```

This code will generate a warning message since the compiler assumes that you did not intend to hide the name and that it was probably by mistake. Note that it is just a warning and not an error. The warning caused by hiding an inherited name can be eliminated through use of the new modifier as shown in the code listed below.

```
class One
{
  public void FunctionOne()
{}
}
```

```
class Two: One
{
  new public void FunctionOne()
{}        // No warning this time
}
```

The new modifier directs the compiler that the name hiding is indeed intended and is not just by mistake. A declaration of a new member hides an inherited member only within the scope of the new member. Consider the code listed below.

```
class One
{
  public static void FunctionOne()
{}
}
class Two : One
{
  new private static void FunctionOne()
{}        // Hides One.FunctionOne in Two only
}
class Three: Two
{
  static void FunctionTwo() { FunctionOne(); }
// Invokes One.FunctionOne
}
```

In the example above, the declaration of FunctionOne in Two hides the FunctionONe that was inherited from One, but since the new FunctionOne in Two has private access, its scope does not extend to Three. Thus, the call FunctionOne() in Three.FunctionTwo is valid and will invoke One.FunctionOne.

Inheritance & Constructors

Inheritance does not allow constructors and destructors of the base class to be inherited by the derived class. However when we create an object of the derived class, the derived class constructor implicitly call the base class default constructor. Consider the following code.

```
using System;
class BaseClass
{
  public BaseClass()
  {
    Console.WriteLine("Base class default constructor");
  }
}
class DerivedClass : BaseClass
{
}

class TestClass
{
  public static void Main()
  {
    DerivedClass d1 =new DerivedClass();
    // Displays 'BaseClass default constructor'
  }
}
```

Note that the DerivedClass constructor can call only the default constructor of BaseClass explicitly. But they can call any BaseClass constructor explicitly by using the keyword base as shown in the code listed below.

```
using System;
class BaseClass
{

public Base()
{
  Console.WriteLine("BaseClass constructor One");
}
public BaseClass(int x)
{
  Console.WriteLine("Base constructor Two");
}
}
class DerivedClass : BaseClass
{
  public DerivedClass() : BaseClass(10)
  // implicitly call the BaseClass(int x)
  {
    Console.WriteLine("DerivedClass constructor");
  }
}
class TestClass
{
public static void Main()
{
  DerivedClass d1 = new DerivedClass();
  // Displays 'BaseClass constructor Two followed by 'DerivedClass
Constructor''
}
}
```

Student Activity 2

1. Explain how inheritance may lead to name hiding. What are the rules that govern name hiding during inheritance?
2. How Constructor behave in context with inheritance. Explain with the help of suitable example.

7.3 POLYMORPHISM

Polymorphism is the term derived from two greek words poly meaning many and morph meaning forms. Thus, polymorphism is a property of an object to exhibit multiple form.

In object-oriented paradigm, polymorphism refers to the objects belonging to different classes respond to the same function call but in different forms. For example, View function takes an argument and displays it on the screen.

- View("Hello there");
- View(46542.93);
- View(1234);

Therefore when the first method is called it will display "Hello there" on the screen. When the second method is called by passing float argument it will display 46542.93 and on the execution of the third method it will display 1234, which is an integer value. Here we can say that the function View is polymorphic.

7.4 OPERATOR OVERLOADING

Normally an operator has a predefined meaning. The number of operands, their types and the computed result are all predefined for each operator. However, additional meanings can also be attached to the existing functions of an operator. Assigning additional meaning to an operator is called operator overloading.

In C# polymorphism is achieved by operator overloading. User-defined operator implementations always take precedence over predefined operator implementations: Only when no applicable user-defined operator implementations exist will the predefined operator implementations be considered.

A class or struct member declaration cannot introduce a member by the same name as the class or struct. A class, struct, or interface permits the declaration of overloaded methods and indexers. A class or struct furthermore permits the declaration of overloaded constructors and operators. For instance, a class, struct, or interface may contain multiple method declarations with the same name, provided these method declarations differ in their signature.

Not all operators are allowed to be overloaded. The overloadable unary operators are:

```
+ - ! ~ ++ -- true false
```

The overloadable binary operators are:

```
+ - * / % & | ^ << >> == != > < >= <=
```

All the rest of the operators cannot be overloaded. In particular, it is not possible to overload member access, method invocation, or the =, &&, ||, ?:, new, typeof, sizeof, and is operators. When a binary operator is overloaded, the corresponding assignment operator is also implicitly overloaded. For example, an overload of operator * is also an overload of operator *=.

Constructors are often overloaded. Overloading of constructors permits a class or struct to declare multiple constructors, provided the signatures of the constructors are all unique. Overloading of indexers permits a class, struct, or interface to declare multiple indexers, provided the signatures of the indexers are all unique.

Operator itself (=) cannot be overloaded. An assignment always performs a simple bit-wise copy of a value into a variable.

In expressions, operators are referenced using operator notation, and in declarations, operators are referenced using functional notation. The following table shows the relationship between operator and functional notations for unary and binary operators. In the first entry, op denotes any overloadable unary operator. In the second entry, op denotes the unary ++ and -- operators. In the third entry, op denotes any overloadable binary operator.

OPERATOR NOTATION	FUNCTIONAL NOTATION
op x	operator op(x)
X op	operator op(x)
X op y	operator op(x, y)

User-defined operator declarations always require at least one of the parameters to be of the class or struct type that contains the operator declaration. Thus, it is not possible for a user-defined operator to have the same signature as a predefined operator.

User-defined operator declarations cannot modify the syntax, precedence, or associatively of an operator. For example, the * operator is always a binary operator, always has the precedence level, and is always left associative.

Overriding Methods

The signature of an inherited method can be modified and the method can be re-implemented in a derived class. The process of re-defining a method is referred to as method-overriding. The override keyword is employed to override a method as shown below.

```
public override void InheritedFunction()
{ ... }
```

The method overrides an inherited virtual method with the same signature when an instance method declaration includes an override modifier. There is a difference between declaring a function virtual and overriding a function. Whereas a virtual method declaration introduces a new method, an override method declaration specializes an existing inherited virtual method by providing a new implementation of the method. Therefore, an override method declaration will report an error if any one of the new, static, virtual, or abstract modifiers is included in the declaration.

Which base function is being overridden in the derived class is determined by examining each base class of the derived class, starting with the direct base class and continuing with each successive direct base class, until an accessible method with the same signature as the overridden method is located. Only the public, protected, protected internal and internal methods declared in the same project as the derived class are considered for the purpose of locating the overridden base method.

Following are some of the rules that govern method overriding. Violating any of them will cause the compiler report an error.

- An overridden base method could not be located.
- The overridden base method must be virtual, abstract, or override method itself.
- The overridden base method cannot be static.
- The override declaration and the overridden base method must have the same declared accessibility.

Consider the following code.

```
class One
{
  int x;
  public virtual void FunctionOne()
{
    Console.WriteLine("x = {0}", x);
  }
}
class Two: One
{
  int y;
  public override void FunctionOne()
{
    base.FunctionOne();
    Console.WriteLine("y = {0}", y);
  }
}
```

Here, the base.FunctionOne() invocation in Two invokes the FunctionOne method declared in One. A base-access disables the virtual invocation mechanism and simply treats the base method as a non-virtual method. Had the invocation in Two been written ((One)this).FunctionOne(), it would recursively invoke the FunctionOne method declared in Two, not the one declared in One.

If a method declaration does not specify override, the method with the same signature as an inherited method will become hidden. Consider the following code.

```
class One
{
  public virtual void FunctionOne()
  {}
}
class Two: One
{
  public virtual void FunctionOne()
  {}   // Warning, hiding inherited FunctionOne()
}
```

Here, the FunctionOne method in Two does not include an override modifier and therefore does not override the FunctionOne method in One. Rather, the FunctionOne method in Two hides the method in One, and a warning is reported because the declaration does not include a new modifier.

Consider the following code.

```
class One
{
  public virtual void FunctionOne()
  {}
}
class Two: One
{
  new private void FunctionOne()
  {}     // Hides One.FunctionOne within Two
}
class Three: Two
{
  public override void FunctionOne()
  {} // No error. overrides One.FunctionOne
}
```

Here, the FunctionOne method in Tow hides the virtual FunctionOne method inherited from One. Since the new FunctionOne in Two has private access, its scope only includes the class body of Two and does not extend to Three. The declaration of FunctionOne in Three is therefore permitted to override the FunctionOne inherited from One.

Student Activity 3

1. What is polymorphism?
2. What do you mean by operator overloading? Explain with the help of suitable example.
3. Explain method-overriding. What are the rules that govern method-overriding?

7.5 SUMMARY

1. Inheritance is one of the primary concepts of any object oriented programming.
2. A class inherits the members of its direct base class.
3. A derived class can hide inherited members by declaring new members with the same name or signature.
4. The constructors and destructors are not inherited to a derived class from a base class.
5. A virtual method in C# specifies an implementation of a method that can be polymorphically overridden derived method.
6. Overloading is the technique to implement the polymorphism.

7. It is possible to omit the keywords virtual from the derived class method or it is possible to declare the derived class method as new.
8. Derived class constructor can call only the default constructor of base class explicitly.

7.6 KEYWORDS

- **Inheritance:** It is a process through which a new class can be created using the features of some other existing classes.
- **Derived Class:** the new class created through inheritance is called as derived class.
- **Base Class:** The class whose features are used in the creation of a new class is called as Base Class.
- **Polymorphism:** It refers to the objects belonging to different classes respond to the same function call but in different forms.
- **Operator Overloading:** To provide multiple functions with the same operators name is called operator overloading.

7.7 REVIEW QUESTIONS

1. Explain various forms of inheritances with suitable examples.
2. What are sealed methods? Where are they used.
3. Illustrate multiple Inheritance with the help of a suitable examples.
4. Differentiate between multiple Inheritance and multilevel Inheritance.
5. Write a few classes, demonstrating polymorphism.
6. Write a program to overload + (Plus) operator to add two complex numbers.
7. Write a program to define a class Automobile which contains number of wheels, engine number and body type. Now inherit a class Santro form automobile class and add the extra features in it.
8. Write a program in C# to demonstrate the concept of method overriding.

7.8 FURTHER READINGS

CLR via C#, Second Edition (Paperback) by Jeffrey Richter, Publisher: Microsoft Press; 2nd edition (February 22, 2006)

Essential C# 2.0 (Microsoft .NET Development Series) (Paperback) by Mark Michaelis, Publisher: Addison-Wesley Professional (July 13, 2006)

C# Programming Language, The (2nd Edition) (Microsoft .NET Development Series) (Hardcover) by Anders Hejlsberg, Scott Wiltamuth, Peter Golde, Publisher: Addison-Wesley Professional; 2 edition (June 9, 2006)

Programming in the Key of C# (Paperback) by Charles Petzold, Publisher: Microsoft Press; 1 edition (July 30, 2003)

UNIT

8

INTERFACES

L E A R N I N G O B J E C T I V E S

After studying this unit, you should be able to

- Define interfaces
- Describe interface properties
- Define interface events
- Discuss interface indexes and implementation

U N I T S T R U C T U R E

8.1 INTRODUCTION

This chapter will introduce you to C# interfaces. An interface merely defines a contract. A class or struct that implements an interface must adhere to its contract. An interface may inherit from multiple base interfaces, and a class or struct may implement multiple interfaces. Interfaces can contain methods, properties, events, and indexers. The interface itself does not provide implementations for the members that it defines. The interface merely specifies the members that must be supplied by classes or struct that implement the interface.

8.2 INTERFACES

At times programmers require to specify what all member must be there, if another programmer re-uses the class. The programmer needs only to specify the members and must leave their implementation to the programmer who would ultimately use the class. To allow such a contract between the programmers the concept of an interface has been introduced. An interface merely defines a contract between the producer and the consumer. The user may create classes and implement one or more interface. A class or struct that implements an interface must adhere to its contract. An interface may inherit from multiple base interfaces, and a class or struct may implement multiple interfaces.

Methods, properties, events, and indexers may be specified in an interface. Note that the interface itself does not provide codes for the members that it defines. It only specifies the members that must be supplied by classes or interfaces that implement the interface.

Interface declarations

In order to create an interface it must be declared. An interface is declared using interface keyword. An interface-declaration consists of an optional set of attributes, followed by an optional set of interface-modifiers, followed by the keyword interface and an identifier that names the interface, optionally followed by an optional interface-base specification, followed by a interface-body, optionally followed by a semicolon. For instance, see the following code.

```
Interface InterfaceOne
{};
```

In the code listed above an interface whose name is InterfaceOne is declared. An interface-declaration may optionally include a sequence of following interface modifiers.

```
new
public
protected
internal
private
```

Note that it is an error for the same modifier to appear multiple times in an interface declaration. The new modifier is only permitted on nested interfaces. It specifies that the interface hides an inherited member by the same name.

The public, protected, internal, and private modifiers control the accessibility of the interface. Depending on the context in which the interface declaration occurs, only some of these modifiers may be permitted.

8.3 BASE INTERFACES

An interface can also inherit from zero or more interfaces, which are called the explicit base interfaces of the interface. When an interface has more than zero explicit base interfaces then in the declaration of the interface, the interface identifier is followed by a colon and a comma-separated list of base interface identifiers.

```
Interface InterfaceTwo : InterfaceOne
{};
```

The explicit base interfaces of an interface must be at least as accessible as the interface itself. For example, it is an error to specify a private or internal interface in the interface-base of a public interface.

It is an error for an interface to directly or indirectly inherit from itself.

The base interfaces of an interface are the explicit base interfaces and their base interfaces. In other words, the set of base interfaces is the complete transitive closure of the explicit base interfaces, their explicit base interfaces, and so on. Consider the following code.

```
interface InterfaceOne
{
  void FunctionOne();
}
interface InterfaceTwo : InterfaceOne
{
  void FunctionTwo(string text);
}
interface InterfaceThree : InterfaceOne
{
  void FunctionThree(string[] items);
```

```
}
interface InterfaceFour : InterfaceTwo, InterfaceThree
{}
```

Here the base interfaces of InterfaceFour are InterfaceOne, InterfaceTwo, and InterfaceThree. An interface inherits all members of its base interfaces.

The interface-body of an interface defines the members of the interface. The members of an interface are the members inherited from the base interfaces and the members declared by the interface itself.

An interface declaration may declare zero or more members. The members of an interface must be methods, properties, events, or indexers. An interface cannot contain constants, fields, operators, constructors, destructors, static constructors, or types, nor can an interface contain static members of any kind.

All interface members implicitly have public access. It is an error for interface member declarations to include any modifiers. In particular, interface members cannot be declared with the abstract, public, protected, internal, private, virtual, override, or static modifiers.

Consider the following code.

```
public delegate void StringListEvent(IStringList sender);
public interface IStringList
{
  void Add(string s);
  int Count { get; }
  event StringListEvent Changed;
  string this[int index] { get; set; }
}
```

The code declares an interface that contains one each of the possible kinds of members: A method, a property, an event, and an indexer.

An interface-declaration creates a new declaration space, and the interface-member-declarations immediately contained by the interface-declaration introduce new members into this declaration space. The following rules apply to interface-member-declarations:

- The name of a method must differ from the names of all properties and events declared in the same interface. In addition, the signature of a method must differ from the signatures of all other methods declared in the same interface.
- The name of a property or event must differ from the names of all other members declared in the same interface.
- The signature of an indexer must differ from the signatures of all other indexers declared in the same interface.

The inherited members of an interface are specifically not part of the declaration space of the interface. Thus, an interface is allowed to declare a member with the same name or signature as an inherited member. When this occurs, the derived interface member is said to hide the base interface member. Hiding an inherited member is not considered an error, but it does cause the compiler to issue a warning. To suppress the warning, the declaration of the derived interface member must include a new modifier to indicate that the derived member is intended to hide the base member.

If a new modifier is included in a declaration that doesn't hide an inherited member, a warning is issued to that effect. This warning is suppressed by removing the new modifier.

8.4 INTERFACE METHODS

Interface methods are declared using interface-method-declarations which can have attributes, return-type and formal parameter list. The attributes, return-type, and formal-parameter-list of an interface method declaration have the same meaning as those of a method declaration in a class.

An interface method declaration is not permitted to specify a method body, and the declaration therefore always ends with a semicolon.

8.5 INTERFACE PROPERTIES

Interface properties are declared using interface-property-declarations. The attributes, type, and identifier of an interface property declaration have the same meaning as those of a property declaration in a class.

The accessors of an interface property declaration correspond to the accessors of a class property declaration, except that the accessor body must always be a semicolon. Thus, the accessors simply indicate whether the property is read-write, read-only, or write-only.

Properties are a new language feature introduced with C#. They provide the opportunity to protect a field in a class by reading and writing to it through the property. C# properties enable this type of protection while also letting you access the property just like it was a field. Let's take a look at how to provide field encapsulation by traditional methods.

```
using System;
public class PropertyHolder
{
    private int someProperty = 0;
    public int getSomeProperty()
    {
        return someProperty;
    }
    public void setSomeProperty(int propValue)
    {
        someProperty = propValue;
    }
}
public class PropertyTester
{
    public static int Main(string[] args)
    {
    PropertyHolder propHold = new PropertyHolder();
    propHold.setSomeProperty(5);
     Console.WriteLine("Property Value: {0}", propHold.getSomeProperty());
     return 0;
    }
}
```

The propertyHoder class has the field we're interested in accessing. It has two methods, getSomeProperty and setSomeProperty. The getSomeProperty method returns the value of the someProperty field. The setSomeProperty method sets the value of the someProperty field.

The PropertyTester class uses the methods of the PropertyHolder class to get the value of the someProperty field in the PropertyHolder class. The Main method instantiates a new PropertyHolder object, propHold. Next it sets the someMethod of propHold to the value 5 by using the setSomeProperty method. Then the program prints out the property value with a Console.WriteLine method call. The argument used to obtain the value of the property is a call to the getSomeProperty method of the propHold object. It prints out "Property Value: 5" to the console.

This method of accessing information in a field has been good because it supports the object-oriented concept of encapsulation. If the implementation of someProperty changed from an int type to a byte type, this would still work.

The same thing can be accomplished much smoother with properties.

```
using System;

public class PropertyHolder
{
    private int someProperty = 0;

    public int SomeProperty
    {
        get
        {
            return someProperty;
        }
        set
        {
            someProperty = valued;
        }
    }
}

public class PropertyTester
{
    public static int Main(string[] args)
    {
       PropertyHolder propHold = new PropertyHolder();

       propHold.SomeProperty = 5;

      Console.WriteLine("Property Value: {0}", propHold.SomeProperty);

       return 0;
    }
}
```

The program listed above shows how to create and use a property. The PropertyHolder class has the "SomeProperty" property implementation. Notice that the first letter of the first word is capitalized. That's the only difference between the names of the property "SomeProperty" and the field "someProperty". The property has two accessors, get and set. The get accessor returns the value of the someProperty field.

The set accessor sets the value of the someProperty field with the contents of "valued".

The PropertyTester class uses the SomeProperty property in the PropertyHolder class. The first line of the Main method creates a PropertyHolder object named propHold. Next the value of the someProperty field of the propHold object is set to 5 by using the SomeProperty property. It's that simple -- just assign the value to the property as if it were a field.

After that, the Console.WriteLine method prints the value of the someProperty field of the propHold object. It does this by using the SomeProperty property of the propHold object. Again, it's that simple -- just use the property as if it were a field itself.

ReadOnly Property

Properties can be made read-only. This is accomplished by having only a get accessor in the property implementation. Since there is no set accessor provided for the property, its value cannot be written.

```
using System;

public class PropertyHolder
{
    private int someProperty = 0;

    public PropertyHolder(int propVal)
    {
        someProperty = propVal;
    }

    public int SomeProperty
    {
        get
       {
            return someProperty;
        }
    }
}

public class PropertyTester
{
    public static int Main(string[] args)
    {
       PropertyHolder propHold = new PropertyHolder(5);

      Console.WriteLine("Property Value: {0}", propHold.SomeProperty);

        return 0;
    }
}
```

The PropertyHolder class has a SomeProperty property that only implements a get accessor. It leaves out the set accessor. This particular PropertyHolder class has a constructor which accepts an integer parameter.

The Main method of the PropertyTester class creates a new PropertyHolder object named propHold. The instantiation of the propHold object uses the constructor of the PropertyHolder that takes an int parameter. In this case, it's set to 5. This initializes the someProperty field of the propHold object to 5. Since the SomeProperty property of the PropertyHolder class is read-only, there is no other way to set the value of the someProperty field. If you inserted PropHold.SomeProperty = 7" into the listing, the program would not compile, because SomeProperty is read-only. When the SomeProperty property is used in the Console.WriteLine method, it works fine. This is because it's a read operation which only invokes the get accessor of the SomeProperty property.

Write-Only Property

Similar to read-only, write-only properties do not implement get accessor for that property. Consider the following program wherein a write-only property has been implemented.

```
using System;

public class PropertyHolder
{
    private int someProperty = 0;
    public int SomeProperty
    {
        set
       {
```

```
            someProperty = value;
           Console.WriteLine("someProperty is equal to {0}", someProperty);
             }
        }
}

public class PropertyTester
{
    public static int Main(string[] args)
    {
        PropertyHolder propHold = new PropertyHolder();
        propHold.SomeProperty = 5;
         return 0;
    }
}
```

The above program shows how to create and use a write-only property. This time the get accessor is removed from the SomeProperty property of the PropertyHolder class. The set accessor has been added, with a bit more logic. It prints out the value of the someProperty field after it's been modified.

The Main method of the PropertyTester class instantiates the PropertyTester class with a default constructor. Then it uses the SomeProperty property of the propHold object to set the someProperty field of the propHold object to 5. This invokes the set accessor of the propHold object, which sets the value of it's someProperty field to 5 and then prints "someProperty is equal to 5" to the console.

8.6 INTERFACE EVENTS

Interfaces can also define events. Interface events are declared using interface-event-declarations. The attributes, type, and identifier of an interface event declaration have the same meaning as those of an event declaration in a class.

8.7 INTERFACE INDEXERS

Indexers can also be members of an interface. Interface indexers are declared using interface-indexer-declarations. The attributes, type, and formal-parameter-list of an interface indexer declaration have the same meaning as those of an indexer declaration in a class.

The accessors of an interface indexer declaration correspond to the accessors of a class indexer declaration, except that the accessor body must always be a semicolon. Thus, the accessors simply indicate whether the indexer is read-write, read-only, or write-only.

Accessing interface members

Interface members are accessed through member access and indexer access expressions of the form I.M and I[A], where I is an instance of an interface type, M is a method, property, or event of that interface type, and A is an indexer argument list.

For interfaces that are strictly single-inheritance (each interface in the inheritance chain has exactly zero or one direct base interface), the effects of the member lookup, method invocation, and indexer access rules are exactly the same as for classes and structs: More derived members hide less derived members with the same name or signature. However, for multiple-inheritance interfaces, ambiguities can occur when two or more unrelated base interfaces declare members with the same name or signature. This section shows several examples of such situations. In all cases, explicit casts can be included in the program code to resolve the ambiguities.

Consider the following code.

```
interface InterfaceOne
{
  int Count { get; set; }
}
interface InterfaceTwo
{
  void Count(int i);
}
interface InterafceThree : InterfaceOne, InterfaceTwo
{}
class ClassOne
{
  void Test(InterfaceTwo x)
{
    x.Count(1);
// Error, Count is ambiguous
  x.Count = 1;
// Error, Count is ambiguous
  ((InterfaceTwo)x).Count = 1;
// Ok, invokes InterfaceTwo.Count.set
  ((InterfaceOne)x).Count(1);
// Ok, invokes InterfaceOne.Count
  }
}
```

Here, the first two statements cause compile-time errors because the member lookup of Count in InterfaceTwo is ambiguous. As illustrated by the example, the ambiguity is resolved by casting x to the appropriate base interface type. Such casts have no run-time costs-they merely consist of viewing the instance as a less derived type at compile-time.

Consider the following code.

```
interface IInteger
{
  void Add(int i);
}
interface IDouble
{
  void Add(double d);
}
interface INumber: IInteger, IDouble {}
class C
{
  void Test(INumber n) {
    n.Add(1);       // Error, both Add methods are applicable
    n.Add(1.0);       // Ok, only IDouble.Add is applicable
    ((IInteger)n).Add(1);  // Ok, only IInteger.Add is a candidate
    ((IDouble)n).Add(1);   // Ok, only IDouble.Add is a candidate
  }
}
```

Here, the invocation n.Add(1) is ambiguous because a method invocation requires all overloaded candidate methods to be declared in the same type. However, the invocation n.Add(1.0) is permitted because only IDouble.Add is applicable. When explicit casts are inserted, there is only one candidate method, and thus no ambiguity.

Consider the following code.

```
interface InterfaceOne
{
  void FunctionOne(int i);
}
interface InterfaceTwo : InterfaceOne
{
  new void FunctionOne(int i);
}
interface InterfaceThree: InterfaceOne
{
  void FunctionTwo();
}
interface InterfaceFour : InterfaceTwo, InterfaceTHree
{}
class ClassOne
{
  void Test(InterfaceFour d)
{
    d.FunctionOne(1);
// Invokes InterfaceTwo.FunctionOne

((InterfaceOne)d).FunctionOne(1);
// Invokes InterfaceOne.FunctionOne

((InterfaceTwo)d).FunctionOne(1);
// Invokes InterfaceTwo.FunctionOne

((InterfaceThree)d).FunctionOne(1);
// Invokes InterfaceOne.FunctionOne
}
}
```

Here, the InterfaceOne.FunctionOne member is hidden by the InterfaceTwo.FunctionOne member. The invocation d.FunctionOne(1) thus selects InterfaceTwo.FunctionOne, even though InterfaceOne.FunctionOne appears to not be hidden in the access path that leads through InterfaceThree.

Student Activity 1

1. What is an interface? What purposes does it serve in a C# program?
2. What are the differences between an interface and a class?
3. What could be the members of an interface? What are the rules that apply to interface member declarations?
4. What purpose does C# properties serve?
5. Using a suitable example, explain how read-only properties may be used.

8.8 INTERFACE IMPLEMENTATIONS

Interfaces are codeless entities and hence cannot be instantiated as classes. However, interfaces may be implemented by classes and structs. To indicate that a class or struct implements an

interface, the interface identifier is included in the base class list of the class or struct. Consider the following code.

```
interface InterfaceOne
{
  object ObjectOne();
}
interface InterfaceTwo
{
  int CompareTo(object other);
}
class ClassOne: InterfaceOne, InterfaceTwo
{
  public object ObjectOne() {…}
  public int CompareTo(object other) {…}
}
```

A class or struct that implements an interface also implicitly implements all of the interface's base interfaces. This is true even if the class or struct doesn't explicitly list all base interfaces in the base class list.

8.9 INTERFACE MAPPING

Note that a class or struct must provide implementations of all members of the interfaces that are listed in the base class list of the class or struct. The process of locating implementations of interface members in an implementing class or struct is known as interface mapping.

Interface mapping for a class or struct locates an implementation for each member of each interface specified in the base class list. The implementation of a particular interface member is determined by examining each class or struct starting with the current class and repeating for each successive base classes, until a match is located.

For purposes of interface mapping, a class member A matches an interface member B when:

- A and B are methods, and the name, type, and formal parameter lists of A and B are identical.
- A and B are properties, the name and type of A and B are identical, and A has the same accessors as B (A is permitted to have additional accessors if it is not an explicit interface member implementation).
- A and B are events, and the name and type of A and B are identical.
- A and B are indexers, the type and formal parameter lists of A and B are identical, and A has the same accessors as B (A is permitted to have additional accessors if it is not an explicit interface member implementation).

Notable implications of the interface mapping algorithm are:

- Explicit interface member implementations take precedence over other members in the same class or struct when determining the class or struct member that implements an interface member.
- Private, protected, and static members do not participate in interface mapping.

Consider the following code.

```
interface IinterfaceOne
{
  object Clone();
}
class ClassOne : InterfaceOne
{
  object InterfaceOne.Clone() {…}
  public object Clone() {…}
}
```

Here, the InterfaceOne.Clone member of ClassOne becomes the implementation of Clone in InterfaceOne because explicit interface member implementations take precedence over other members.

If a class or struct implements two or more interfaces containing a member with the same name, type, and parameter types, it is possible to map each of those interface members onto a single class or struct member. Consider the following code.

```
interface InterfaceOne
{
  void Paint();
}
interface InterfaceTwo
{
  void Paint();
}
class CkassOne : InterfaceOne, InterfaceTwo
{
  public void Paint() {…}
}
```

Here, the Paint methods of both the interfaces are mapped onto the Paint method in ClassOne. It is of course also possible to have separate explicit interface member implementations for the two methods.

If a class or struct implements an interface that contains hidden members, then some members must necessarily be implemented through explicit interface member implementations. Consider the following code.

```
interface InterfaceOne
{
  int P { get; }
}
interface InterafceTwo: InterfaceOne
{
  new int P();
}
```

An implementation of this interface would require at least one explicit interface member implementation, and would take one of the following forms

```
class ClassOne : InterfaceTwo
{
  int InterfaceOne.P { get {…} }
  int InterfaceTwo.P() {…}
}
class ClassOne: InterfaceTwo
{
  public int P { get {…} }
  int InterfaceTwo.P() {…}
}
class ClassOne: InterfaceTwo
{
  int InterfaceOne.P { get {…} }
  public int P() {…}
}
```

When a class implements multiple interfaces that have the same base interface, there can be only one implementation of the base interface. Consider the following code.

```
interface IControl
{
  void Paint();
}
interface ITextBox: IControl
{
  void SetText(string text);
}
interface IListBox: IControl
{
  void SetItems(string[] items);
}
class ComboBox: IControl, ITextBox, IListBox
{
  void IControl.Paint() {…}
  void ITextBox.SetText(string text) {…}
  void IListBox.SetItems(string[] items) {…}
}
```

As is clear, it is not possible to have separate implementations for the IControl named in the base class list, the IControl inherited by ITextBox, and the IControl inherited by IListBox. Indeed, there is no notion of a separate identity for these interfaces. Rather, the implementations of ITextBox and IListBox share the same implementation of IControl, and ComboBox is simply considered to implement three interfaces, IControl, ITextBox, and IListBox.

The members of a base class participate in interface mapping. Consider the following code.

```
interface Interface1
{
  void FunctionOne();
}
class Class1
{
  public void FunctionOne() {}
  public void FunctionTwo() {}
}
class Class2: Class1, Interface1
{
  new public void FunctionTwo() {}
}
```

Here, the method FunctionOne in Class1 is used in Class2's implementation of Interface1.

As it is compulsory for a class or a struct which implements an interface to implement all the methods and properties of the interface this is called interface mapping the signature and the arguments of all the members of the interface should map with that declared in the class.

Consider the following code.

```
interface Print
{
void setText(int number);
string getText();
}

public class Student: Print
{
  string name[ ]= new string[];
```

```
    public void setText( int e)
    {
         for( int a=0; a<e;a++)
         {
           System.Console.WriteLine("enter the name");
           Name[a]=System.Console.ReadLine();
       }
  }
  public string getText(int f)
    {
      return name[f];
    }
    public static void Main()
    {
      System.Console.WriteLine("enter the number of students you want to
      enter the name");
    int e=System.Console.ReadLine();
    Student s= new Student();
    s.setText(e);
    for(a=0;a<e;a++)
    System.Console.WriteLine("The name of student is:"+  s.getText(a));
    Teacher t=new Teacher()
    System.Console.WriteLine("Enter the number of subjects taken by the
    teacher");
    int h=System.Console.ReadLine();
    t.setText(h);
    }
  }
  class Teacher: Print
  {
    public void setText (int subject)
    {
      System.Console.WriteLine("This teacher takes "+subject+" subjects");
    }
    public string getText(){}
  }
```

the output would be:

```
enter the number of students you want to enter the name:
3
enter the name:
Anita
enter the name
Rekha
Enter the name:
Sulekha

The name of the student is Anita
The name of the student is Rekha
The name of the student is Sulekha
Enter the number of subjects taken by the teacher:
5
This teacher takes 5 subjects
```

In the above program we have used two different classes Student and Teacher, you will see that we have mapped both the methods of the interface Print in the classes as it implements the interface but the function performed by the methods are different in both the classes. It is not

important to have the same definition or the function performed by the method in various classes, which implement the method. Also note that in the class Teacher the body of getText() method is not there because we are not going to do any function but we still have to map it and declare with empty body.

Interface implementation inheritance

A class inherits all interface implementations provided by its base classes.

Without explicitly re-implementing an interface, a derived class cannot in any way alter the interface mappings it inherits from its base classes. For example, in the declarations

```
interface IControl
{
  void Paint();
}
class Control: IControl
{
  public void Paint() {…}
}
class TextBox: Control
{
  new public void Paint() {…}
}
```

the Paint method in TextBox hides the Paint method in Control, but it does not alter the mapping of Control.Paint onto IControl.Paint, and calls to Paint through class instances and interface instances will have the following effects

```
Control c = new Control();
TextBox t = new TextBox();
IControl ic = c;
IControl it = t;
c.Paint();      // invokes Control.Paint();
t.Paint();      // invokes TextBox.Paint();
ic.Paint();     // invokes Control.Paint();
it.Paint();     // invokes Control.Paint();
```

However, when an interface method is mapped onto a virtual method in a class, it is possible for derived classes to override the virtual method and alter the implementation of the interface. For example, rewriting the declarations above to

```
interface IControl
{
  void Paint();
}
class Control: IControl
{
  public virtual void Paint() {…}
}
class TextBox: Control
{
  public override void Paint() {…}
}
the following effects will now be observed
Control c = new Control();
TextBox t = new TextBox();
IControl ic = c;
IControl it = t;
c.Paint();      // invokes Control.Paint();
t.Paint();      // invokes TextBox.Paint();
```

```
ic.Paint();      // invokes Control.Paint();
it.Paint();      // invokes TextBox.Paint();
```

Since explicit interface member implementations cannot be declared virtual, it is not possible to override an explicit interface member implementation. It is however perfectly valid for an explicit interface member implementation to call another method, and that other method can be declared virtual to allow derived classes to override it. Consider the following code.

```
interface IControl
{
  void Paint();
}
class Control: IControl
{
  void IControl.Paint() { PaintControl(); }
  protected virtual void PaintControl() {…}
}
class TextBox: Control
{
  protected override void PaintControl() {…}
}
```

Here, classes derived from Control can specialize the implementation of IControl.Paint by overriding the PaintControl method.

Ambiguity in the methods of the interface

Consider the following code.

```
public interface Home
{
void address();
}
public interface Office
{
void address(string name);

}
interface Employee: Home, Office {}

class company
{
  void employee(Employee e)
  {
  e.address("New Delhi");
e.address("Mumbai");
}
  public static void Main()
  {
  company c=new company();
  c.employee();
  }
}
```

The output would be that it would give an error saying that the method is ambiguous because it implements interface Home and Office both of which have same method address. Therefore it won't know which method to use.

The improved program would be to explicitly specify which method to take from which interface.

```
public interface Home
{
void address();
}
public interface Office
{
void address(string name);

}
interface Employee: Home, Office {}

class company
{
  void employee(Employee e)
  {
  (( Home ) e ).address("New Delhi");
(( Office ) e).address("Mumbai");
System.Console.WriteLine("The home address is:" + ((Home) e).address));
System.Console.WriteLine("The office address is:"+((Office)e).address));
}
  public static void Main()
  {
  company c=new company();
  c.employee();
  }
}
The output would be
The home address is:   New Delhi
The office address is :     Mumbai
```

Now in this case we have specified the method to be taken from which interface.

Student Activity 2

1. How are interfaces implemented? Explain giving suitable examples.
2. What is interface mapping?

8.10 INTERFACE RE-IMPLEMENTATION

A class that inherits an interface implementation is permitted to re-implement the interface by including it in the base class list.

A re-implementation of an interface follows exactly the same interface mapping rules as an initial implementation of an interface. Thus, the inherited interface mapping has no effect whatsoever on the interface mapping established for the re-implementation of the interface. Consider the following code.

```
interface IControl
{
  void Paint();
}
class Control: IControl
{
  void IControl.Paint() {…}
}
```

```
class MyControl: Control, IControl
{
  public void Paint() {}
}
```

The fact that Control maps IControl.Paint onto Control.IControl.Paint doesn't affect the re-implementation in MyControl, which maps IControl.Paint onto MyControl.Paint.

Inherited public member declarations and inherited explicit interface member declarations participate in the interface mapping process for re-implemented interfaces. Consider the following code.

```
interface IMethods
{
  void FunctionOne();
  void FunctionTwo();
  void FunctionThree();
  void FunctionFour();
}
class Base: IMethods
{
  void IMethods.FunctionOne() {}
  void IMethods.FunctionTwo() {}
  public void FunctionThree() {}
  public void FunctionFour() {}
}
class Derived: Base, IMethods
{
  public void FunctionOne() {}
  void IMethods.FunctionTHree() {}
}
```

Here, the implementation of IMethods in Derived maps the interface methods onto Derived.FunctionOne, Base.IMethods.FunctionTwo, Derived.IMethods. FunctionThree, and Base.FunctionFour.

When a class implements an interface, it implicitly also implements all of the interface's base interfaces. Likewise, a re-implementation of an interface is also implicitly a re-implementation of all of the interface's base interfaces. Consider the following code.

```
interface IBase
{
  void FunctionOne();
}
interface IDerived: IBase
{
  void FunctionTwo();
}
class C: IDerived
{
  void IBase.FunctionOne() {…}
  void IDerived.FunctionTwo() {…}
}
class D: C, IDerived
{
  public void FunctionOne() {…}
  public void FunctionTwo() {…}
}
```

Here, the re-implementation of IDerived also re-implements IBase, mapping IBase.FunctionOne onto D.FunctionOne.

Abstract classes and interfaces

Like a non-abstract class, an abstract class must provide implementations of all members of the interfaces that are listed in the base class list of the class. However, an abstract class is permitted to map interface methods onto abstract methods. Consider the following code.

```
interface IMethods
{
  void FunctionOne();
  void FunctionTwo();
}
abstract class C: IMethods
{
  public abstract void FunctionOne();
  public abstract void FunctionTwo();
}
```

Here, the implementation of IMethods maps FunctionOne and FunctionTwo onto abstract methods, which must be overridden in non-abstract classes that derive from C.

Note that explicit interface member implementations cannot be abstract, but explicit interface member implementations are of course permitted to call abstract methods. Consider the following code.

```
interface IMethods
{
  void FunctionOne();
  void FunctionTwo();
}
abstract class C: IMethods
{
  void IMethods.FunctionOne() { FF(); }
  void IMethods.FunctionTwo() { GG(); }
  protected abstract void FF();
  protected abstract void GG();
}
```

Here, non-abstract classes that derive from C would be required to override FF and GG, thus providing the actual implementation of IMethods.

Student Activity 3

1. Explain the concept of interface reimplementation with the help of suitable examples.
2. Compare and contrast between an interface and an abstract class.

8.11 SUMMARY

- An Interface merely defines a contract between the producer and the consumer.
- Interface can contain methods, properties, events, and indexers.
- An interface can inherit from zero or more interface, which are called the explicit base interfaces of the interface.
- It is an error for an interface to directly or indirectly inherit from itself.
- Interfaces may be implemented by classes and structs.
- A class or struct that implements an interface also implicitly implements all of the interface's base interface.
- A class that inherits an interface implementation is permitted to reimplement the interface by including it in the base class list.

- Like a non-abstract class, an abstract class must provide implementation of all members of the interface that are listed in the base class.
- It is compulsory for a class or a struct, which implements an interface to implements all the methods and properties of the interface.
- Without explicitly reimplementing an interface, a derived class cannot in any way alter the interface mappings it inherits from its base classes.

8.12 KEYWORDS

- **Interface:** An interface contains only the signatures of methods, delegates or events. The implementation of the methods is done in the class that implements the interface
- **Interface Mapping:** The process of locating implementations of interface members in an implementing class or struct is known as interface mapping.
- **Interface Reimplementation:** A class that inherits an interface implementation is permitted to re-implement the interface by including it in the base class.
- **Abstract Class:** A class with the prefix word abstract is called as abstract class. Abstract class cannot be instantiated.

8.13 REVIEW QUESTIONS

1. What is an interface? What purposes does it serve in a C# program?
2. How is multiple inheritances implemented using interfaces?
3. What is interface mapping?
4. Using a suitable example, explain how read-only properties may be used.
5. Compare and contrast between an interface and an abstract class.
6. What do you mean by multiple inheritance and multilevel inheritance? Differentiate between them.
7. Write a program to define a structure and implement interface.
8. Write a program to implement the concept of multilevel interfaces.

8.14 FURTHER READINGS

CLR via C#, Second Edition (Paperback) by Jeffrey Richter, Publisher: Microsoft Press; 2nd edition (February 22, 2006)

Essential C# 2.0 (Microsoft .NET Development Series) (Paperback) by Mark Michaelis, Publisher: Addison-Wesley Professional (July 13, 2006)

C# Programming Language, The (2nd Edition) (Microsoft .NET Development Series) (Hardcover) by Anders Hejlsberg, Scott Wiltamuth, Peter Golde Publisher: Addison-Wesley Professional; 2 edition (June 9, 2006)

Programming in the Key of C# (Paperback) by Charles Petzold, Publisher: Microsoft Press; 1 edition (July 30, 2003)

UNIT

9

CONFIGURATION AND DEPLOYMENT

L E A R N I N G O B J E C T I V E S

After studying this unit, you should be able to

- Define preprocessor
- Know about preprocessor identifier and expression
- Define term documentation
- Describe .Net components

U N I T S T R U C T U R E

9.1 INTRODUCTION

This chapter will introduce you to preprocessing directives in C#. C# preprocessor is fundamentally very similar to C preprocessor but in C# only concept has been taken from C. C# compiler does not have a separate preprocessor, the directives described in this C# are processed as if there was one. You will also learn about the various steps while documenting and how if you follow the basic rules and standards provided in C#, you can save a lot of time while documenting. Later in the chapter you will learn about .NET components, interoperability between .NET components and COM and NGSW components, which is a set of new services, software and solutions for Microsoft customers, built around the Internet, Windows and new devices.

9.2 PRE-PROCESSOR

As the term implies, a pre-processor is a utility that processes the source code file(s) before the actual compilation begins. There are a number of actions taken by the pre-processor to produce the output in the desirable format of the compiler. These actions/commands are embedded into the source-code by the programmer using pre-processor directives.

The term "pre-processing directives" in C# is used for consistency with the C programming language only. In C#, there is no separate pre-processing step; pre-processing directives are processed as part of the lexical analysis phase during compilation.

Pre-processing directives always begin with a '#' character, which must be at the beginning of the line, excepting whitespace. Whitespace may optionally occur between the '#' and the following identifier.

```
pp-directive:
pp-declaration
pp-conditional-compilation
pp-line-number
pp-diagnostic-line
pp-region
```

For pp-declaration, pp-conditional-compilation, and pp-line-number directives, the rest of the line is lexically analyzed according the usual rules, and comments and white-space are ignored. It is illegal for an input element (such as a regular comment) to be unterminated at the end of the line.

For pp-diagnostic-line and pp-region directives, the rest of the line is not lexically analyzed.

Pre-processing identifiers

Pre-processing identifiers employ a grammar similar to the grammar used for regular C# identifiers:

```
pp-identifier:
An identifier-or-keyword that is not true or false
```

The symbols true and false are not legal pre-processing identifiers, and so cannot be defined with #define or undefined with #undef.

Pre-processing expressions

The operators !, ==, !=, && and || are permitted in pre-processing expressions. Parentheses can be used for grouping in pre-processing expressions. Pre-processing expressions are evaluated at compile-time according to the same rules as boolean expressions are evaluated at run-time. An identifier that is "defined" evaluates as true, otherwise as false.

```
pp-expression:
pp-equality-expression
pp-primary-expression:
true
false
pp-identifier
(   pp-expression   )
pp-unary-expression:
pp-primary-expression
!   pp-unary-expression
pp-equality-expression:
pp-equality-expression   ==   pp-logical-and-expression
pp-equality-expression   !=   pp-logical-and-expression
pp-logical-and-expression:
pp-unary-expression
pp-logical-and-expression   &&   pp-unary-expression
pp-logical-or-expression:
pp-logical-and-expression
pp-logical-or-expression   ||   pp-logical-and-expression
Pre-processing declarations
```

Names can be defined and undefined for use in pre-processing. A #define defines an identifier within the scope of a file. A #undef "undefines" an identifier within the scope of a file-if the identifier was defined earlier then it becomes undefined. If an identifier is defined then it is semantically equivalent to true; if an identifier is undefined then it is semantically equivalent to false.

Note that defining a pre-processing identifier with #define has no impact on any uses of that identifier outside of pre-processing directives.

pp-declaration:

```
#define pp-identifier  new-line
#undef  pp-identifier  new-line
```

```
The example:
#define A
#undef B
class C
{
#if A
  void F() {}
#else
  void G() {}
#endif
#if B
  void H() {}
#else
  void I() {}
#endif
}
becomes:
class C
{
  void F() {}
  void I() {}
}
```

A pp-declaration is restricted to occur before any tokens in the input file. In other words, #define and #undef must precede any "real code" in the file, or a compile-time error occurs. Thus, it is possible to intersperse #if and #define as in the example below:

```
#define A
#if A
  #define B
#endif
namespace N
{
  #if B
  class Class1 {}
  #endif
}
```

The following example is illegal because a #define follows real code:

```
#define A
namespace N
{
  #define B
  #if B
  class Class1 {}
  #endif
}
```

A #undef may "undefine" a name that is not defined. The example below defines a name and then undefines it twice; the second #undef has no effect but is still legal.

```
#define A
#undef A
#undef A
#if, #elif, #else, #endif
pp-conditional-compilation:
# if   pp-expression   new-line
# elif   pp-expression   new-line
# else   new-line   groupopt
# endif   new-line
```

A set of pp-conditional-compilation directives are used to conditionally include or exclude portions of program text. The pp-conditional-compilation directives must occur in order, as follows: exactly one #if directive, zero or more #elif directives, zero or one #else directive, and exactly one #endif directive. Sets of pp-conditional-compilation directives can nest, as long as one complete set occurs entirely between two directives of the containing set.

When a set of pp-conditional-compilation directives is processed, it causes some of the text between directives to be included, or subject to lexical analysis, and some to be excluded, and not subject to any further processing. The text to be included or excluded is the text that lies strictly between the pp-conditional-compilation directives. The affected text is included or excluded before being subject to lexical analysis; if excluded, it is not scanned for input elements such as comments, literals, or other tokens.

Text is included or excluded according to the results of evaluating the expression(s) in the directives as follows:

- If the pp-expression on the #if directive evaluates to true, then the text between the #if and the next pp-conditional-compilation directive in the set is included, and all other text between the #if and the #endif directive is excluded.
- Otherwise, if any #elif directives are present, then the pp-expressions associated with them are evaluated in order. If any evaluates to true, then the text between the first #elif directive that evaluates to true and the next pp-conditional-compilation directive in the set is included, and all other text between the #if and the #endif directives is excluded.
- Otherwise, if an #else directive is present, then the text between the #else directive and the #endif directive is included, and all other text between the #if and the #endif directives is excluded.
- Otherwise, all of the text between the #if and the #endif directives is excluded.

The example:

```
#define Debug
class Class1
{
#if Debug
  void Trace(string s) {}
#endif
}
becomes:
class Class1
{
  void Trace(string s) {}
}
If sections can nest. Example:
#define Debug   // Debugging on
#undef Trace    // Tracing off
class PurchaseTransaction
{
  void Commit() {
    #if Debug
      CheckConsistency();
      #if Trace
        WriteToLog(this.ToString());
      #endif
    #endif
    CommitHelper();
  }
}
```

Text that is not included is not subject to lexical analysis. For example, the following is legal despite the unterminated comment in the "#else" section:

```
#define Debug   // Debugging on
class PurchaseTransaction
{
  void Commit() {
    #if Debug
      CheckConsistency();
    #else
        /* Do something else
     #endif
  }
}
```

Pre-processing directives are not processed if they appear inside other input elements. For example, the following program:

```
class Hello
{
  static void Main() {
    System.Console.WriteLine(@"hello,
#if Debug
    world
#else
    Nebraska
#endif
      ");
  }
}
produces the following output
hello,
#if Debug
    world
#else
    Nebraska
#endif
#error and #warning
```

The #error and #warning directives enable code to report error and warning conditions to the compiler for integration with standard compile-time errors and warnings.

```
pp-diagnostic-line:
# error   pp-message
# warning   pp-message
```

The example

```
#warning Code review needed before check-in
#define DEBUG
#if DEBUG && RETAIL
  #error A build can't be both debug and retail!
#endif
class Class1
{…}
```

always produces a warning ("Code review needed before check-in"), and produces an error if the pre-processing identifiers DEBUG and RETAIL are both defined.

```
#region and #endregion
```

pp-region:

```
# region   pp-message
# endregion   pp-message
```

The #region and #endregion directives bracket a "region" of code. No semantic meaning is attached to a region; regions are intended for use by the programmer or automated tools to mark a piece of code. The message attached to the #region and #endregion has no semantic meaning; it merely serves to identify the region. Each source file must have an equal number of #region and #endregion directives, and each #region must be matched with a #endregion later in the file.

#region and #endregion must nest properly with respect to conditional compilation directives. More precisely, it is illegal for a pp-conditional-compilation directive to occur between a #region directive and it's matching #endregion directive, unless all of the pp-conditional-compilation directives of the set occur between #region and #endregion.

#line

The #line feature enables a developer to alter the line number and source file names that are used by the compiler in output such as warnings and errors. If no line directives are present then the line number and file name are determined automatically by the compiler. The #line directive is most commonly used in meta-programming tools that generate C# source code from some other text input. After a #line directive, the compiler treats the line after the directive as having the given line number (and file name, if specified).

Student Activity 1

1. What is a pre-processor?
2. What are the C# language's preprocessor directives? What are the main uses of preprocessing directives?

9.3 DOCUMENTATION

Comments and Documentation is one of the most hated tasks by programmers. By using C# programmers can automatically builds the documentation and comments in the code.

The C# compiler checks the comments and generates the XML. It also generates errors if it finds any false tag or false references. This means there are some basic rules and standards to write the comments in C#. If you want to use this C# feature you must need to follow these basic rules and standards. If you follow these standards, it might save your lot of documentation time.

Lets get to the rules and syntax for commenting in C#. You are expected to know a bit of XML before we start. The comments meant for documentation should have /// three slash at the begining of the comment-line. As C# supports C++ style commenting so commenting can be done by // two slash too, but for documentation /// is necessary. Any line starting with /// shal be interpreted as a documenting comment.

Generating documentation requires following steps:

- Describing an Element
- Adding remarks and lists
- Describing Parameters
- Describing Methods/Properties
- Providing Examples
- Compiling the Code

We will go through these steps one by one.

1. Describing an Element

An element of documentation can be described in the source code using <summary> tag.

```
///<summary> description of element </summary>
```

You can add paragraph to the description by using <para> tag. Reference to other element are added using <see> tag.

```
 <para> this uses private variable
<see cref="nFactorial"/>
</para>
```

Beware of not giving wrong reference. C# compiler check each and every reference and will report error if it finds wrong references.

2. Adding Remark and List

Remark is where bulk of the documentations is placed. A remark is inserted using <remarks> tag. This is in contrast with the summary where you should only provide a brief description. You can use para and lists in the remarks section, which can be bulleted or numbered

```
///<list type="bullet">
///<item>Constructor:
///<see cref="Factorial"/>
///</item>
///</list>
```

Another useful tag that is used to reference and describe the parameter passed is <paramref>.

```
///<remarks>
/// <paramref name="nFactorialToComp"/>in the private variable
///<see cref="nFactorialToComp"/>
///</remarks>
```

3. Describing Parameters

The parameter(s) passed to a method mat be decribed using <param> tag.

```
///<param name="nFactorialToComp">
///.........
////</param>
```

You can use <para> tag in between too.

4. Properties

For describing properties of the object etc. <value> tag is used. With this tag you can specifically flag a property. The <value> tag more or less replaces <summary> tag.

5. Example

The <example> tag includes an illustrative example of your application. You can give a complete code with <code> tag that contains a real C# code to show the usage of the application.

```
///<example> this is how u use the code
///<code>
///public static void Main(string[] srgs)
///.......
///</code>
//</example>
```

6. Compling the Code

In order to compile the source code having documentation tags, use /doc switch with the C# compiler. Assuming that the source file name is Factorial.cs, the following is the appropriate command to compile the documentation as well.

```
c:\>csc /doc:Factorial.xml /out:Factorial.exe Factorial.cs
```

After the XML file is generated you will see some constant that you never supplied appearing there. They are the ID's that C# adds for reference and to catagerize members like Methods, Properties, Asemblies etc. Let us have a look at these IDs.

```
N----Namespace
T----Type,This can be Class ,Interface,struct,enum or delegates
```

```
F----fields of class
P---Property indexer or indexed property
M---Methods special methods constructor and operator(overloaded)
E----Events
!----error string (C# was unable to resolve some references)
```

9.4 .NET COMPONENTS

.NET is definitely a great improvement as far as COM technology is considered. It brings us nearer to true code reusability. However, many organizations have invested substantially huge amounts in building COM components. COM has been there in the market for almost 7 years. And, the last decade saw a substantial growth of the IT industry in various segments, which means that there has been really so much of code packed in the form of components. So, it is almost impracticable for one to re-write the entire code into .NET components, however good they may be for designing more robust applications. Existing applications using COM components are already tested, and, if one were to shift to .NET components, this testing would have to be carried out again. So, there is a great need for interoperability between the COM and .NET components.

Managed Environment V/S Unmanaged Environment

Code that runs under .NET Environment (CLR) is said to be managed code. It is so because the CLR takes care about automatic memory management of the code running under its supervision and also provides us with cross language capabilities. The code that runs out of .NET Environment (CLR) is said to be unmanaged code. Such code is usually tied to a platform, and will not acquire the features such as automatic memory management, etc.

.NET Framework permits us to move between managed and unmanaged code. The following summarizes the best of .NET components.

1. XCOPY deployment: Unlike COM, .NET components can be copied to any part of the location and used independently of other versions of the same component. This avoids versioning problems that arise due to entries that are made in Windows Registry. Deploying two versions of the same components sometimes is called as side-by-side deployment.
2. Platform Neutral: .NET components are platform independent.
3. IL Integration: Unlike the binary standard, its lower common denominator is Intermediate Language. You can design the components using Intermediate Language itself. This is what facilitates cross-language capabilities of .NET Framework.

Tools for Interoperability between .NET and COM

Type Library Importer (TlbImp.exe) tool converts the type definitions found within a COM type library into equivalent definition in managed Metadata format. Type Library Exporter (TlbExp.exe) Uses a managed assembly as input, generates a type library containing COM definitions of the public types defined in that assembly.

Assembly Registration tool (RegAsm.exe) enables classic COM clients to call managed classes. RegAsm reads the Metadata within an assembly and then adds the necessary entries to the registry. So classic COM clients can create the managed classes transparently.

ActiveX Control Importer (aximp.exe) uses an ActiveX control's type library as input and generates a wrapper control that allows the ActiveX control to be hosted by Windows Forms.

Converting existing COM Components to .NET

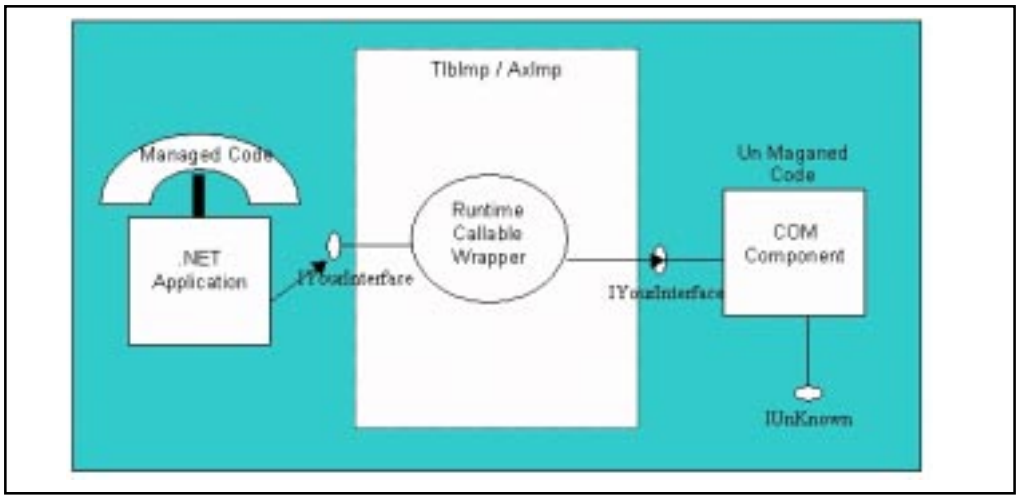
The tool used to covert the existing COM Components to .NET Components is TlbImp, which stands for Type Library Import. The following diagram describes what happens when a COM component is converted to .NET component and how it is called.

Example:

Assume that there is already an existing COM Component called testcom.dll. In order to use this component, you need to first covert the testcom.dll to .NET aware component. You can do so by issuing the following command at the command prompt:

```
TlbImp testcom.dll /out:testnetcom.dll
```

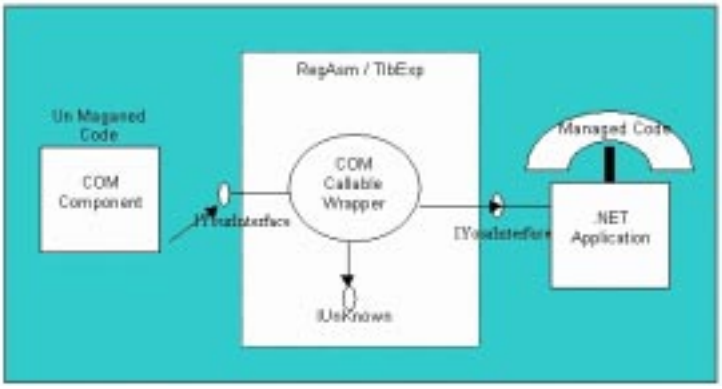
This command will create .NET component from the COM. In order to use newly created .NET component, testnetcom.dll, you will have to reference this DLL during compilation.



Converting Existing ActiveX components to .NET

In order to use an existing ActiveX components in .NET, we need a tool called AxImp that stands for ActiveX import.

Example



named testacitvex.ocx. In order to make this the command prompt.

ivex.dll. In order to use the above files in .NET estactivex.dll and use the control just like any

l Applications

onal components, one should use a tool called llowing diagram explains what exactly happens o COM.

Building a simple .NET component Using C#

The following program in C# demonstrates how to create a .NET component in C#.

```
using System;

class Test
{
public int testmethod()
{
    return 5*5;
}}
```

Note that for a .NET component,

1. The application should not contain Main method
2. Save the file with firstcom.cs
3. Compile by issuing csc /t:library /out:firstcom.dll first.cs

Now your .NET component firstcom.dll is ready.

In order to make it available to traditional applications running on your machine, you should issue the following command:

```
RegAsm firstcom.dll
```

Once you have done that, in order to test it, open Visual Basic and refer to the firstcom.dll and write a simple application. By default, the components designed using C# on Windows NT4.0, are both threaded.

9.5 COM

Component is nothing but the reusable piece of software in binary form that can be plugged into other components from other vendors. Reusable software programs that share a common interface allowing easy integration into applications that is it is a Binary standard that allows any two components to communicate that component may be written in any language.

Need for Component

Today's applications are time consuming to develop, difficult and costly to maintain and we cannot remove a feature, upgraded or replaced with alternatives. The solution for this problem is reusable software components. Each function has a distinct functionality. Microsoft word Word Processing software. Microsoft Excel Spread sheet-analyze &manipulates data. This is a situation where we find ourselves requiring features that are not a part of the application we are using.

Solution

The solution to this problem is to isolate the functionality that is required by many applications and creates a new application from it and makes into a common functionality. Each time the client requires the functionality then it should be called and uses its features as though they were its own. In our project we create the Components from the five applications so that the users can easily use it in many languages and upgrade it easily.

COM Frame Work

To create application that can interact with any other application requires that there be a standard manner in which the two applications will interact. So, for that we need a framework that defines the rules that will enable to interact with each other irrespective of who has created the application.

COM as Blue Print

If all the application follows the same rules for interaction then application created by different people can interact with each other and we can exploit any application to obtain the feature we need.

Microsoft has defined a model that sets standard that will enable application to interact with each other. This model is referred to as the COM. It defines the rules which when implemented will enable applications to interact with other application in a uniform manner. Both calling application and the called application need to follow the rules specified by the COM.

Calling application search the hard disk will be a time consuming affair. Thus COM states all applications that can be provide services to each other application must be registered at a central location. But COM doesn't define what this location is to be or the information that is to be registered. Thus COM acts as blue print. So Com just defines the theory but not how they are to be implemented.

Microsoft's Component Object Model (COM)

When we register a COM or DCOM object within the window registry, we must reference that object's Globally Unique Identifier (GUID), which is a 128-bit (16-byte) number that uniquely identifies the object. Whenever a client program accesses a COM or DCOM object, the client program will use the GUID to reference the object within the windows Registry.

Another powerful COM feature is language independence. COM clients and COM objects have the same layout and behavior at run time, regardless of which language we use to produce the component. COM's independence from any specific language lets build systems with many different components that other programmers create in different language, whether Visual Basic,

C++, or Java. The language we use to create the component simply does not matter in COM.

First, it helps in split up large systems into manageable subsystems early in the design phase. Second, it assists to implement each subsystem with a component that we create with any Com-capable tool. Third, it lets individual teams working on each subsystem has complete.

The two most notable of these technologies are OLE (Object Linking and Embedding) and ActiveX. These both use COM to facilitate all the interaction between objects. OLE uses COM to communicate between applications, allowing users to link or embed parts of one application's data and display into another application.

Interfaces of a Component

The fundamental concept behind both COM and DCOM is the interface. An interface is an agreement between a client and an object about how they will communicate with each other. When we define interfaces in languages other than the visual basic, we have to use Microsoft's Interface Definition Language (IDL), which we must compile within the Microsoft Interface Definition Language Compiler (MIDL).

Globally Unique Identifier (GUID)

Each interface that we define in a COM object will include a Universally Unique Identifier (UUID), which the operating system constructs exactly as it does a Globally Unique Identifier. When client programs access an interface the COM object exposes, the programs will actually reference the interface's UUID. An interface is an agreement between a client and an object about how they communicate. The interface becomes a communication channel between the client and the object. Within our program code, an interface is a collection of related procedures and functions.

Functions of Interfaces

When we create objects that we build around COM and clients to communicate with the COM objects, both the clients and the objects must communicate exclusively through interfaces.

An interface between two objects is actual program code. A logical interface is the model for the program code itself. An interface plays the mediator's role between the client and the object and is a contract that specifies the work the object must do, but it dose nit specify how the object should accomplish the work. In addition, an interface is a communications protocol that defines a set of methods complete with names, arguments and return type. A COM object must implement at lest one interface, although the object is free to implement as many interfaces as it requires. The

object creates an array of function pointers, called a V table (the "V" stands for Virtual), and passes a V Table pointer to the client. The client sees the V Table as the interface and uses the V Table pointer it received from the client to locate a particular function pointer.

When the client finds the function pointer, the client invokes the method directly. In other words, when programs access the COM object, they will always access the V Table and receive a vptr from the V Table's. When we compile the COM-based project, the compiler feeds the Interface Definition Language file to the Microsoft Interface Definition Language (MIDL) compiler, produces a binary description file called a type library. Unlike C++ and Java, Visual Basic does not require using Interface Definition Language or the MIDL compiler. The Visual Basic Interactive Development environment (IDE) creates a type library directly from our Visual Basic source code and builds the type library information. If we want to see the Interface Definition Language for a particular COM component, we can use the ole view, utility to "reverse engineer" a type library into Interface Definition Language text. The type library lets development tools, such as the Visual Basic compiler and the Visual J++ compiler, build the Table binding at compile time.

Component Creation in C#

Interoperability

1. C# includes native support for the COM and windows based applications.
2. Allowing restricted use of native pointers.
3. Users no longer have to explicitly implement the unknown and other COM interfaces, those features are built in.
4. C# allows the users to use pointers as unsafe code blocks to manipulate your old code.
5. Components from VB NET and other managed code languages and directly be used in C#.

Compiling the Component

The creation of the component is a simply writing a class program with in a normal application but in compilation we have to create a library instead of an application. After creating a separate library file it can be used in a client application.

The compilation process should be as follows

```
csc  /r:  REFRENCE DLL /t  library /out DLLNAME.DLL Program name.cs
The "/t library" in compilation code says the C# compiler to create a
library and also tells
```

to not to look for a 'static Main() method. In "/r" we have to add the required namespace DLL. After the compilation we have the required library which is ready to use in the client application. To use a component in client application we have to compile like this

```
csc \r:components library DLL name   program name.cs
Ex:  csc /r: arun.dll component.cs
```

The code is simple, and there are only two files (client and server) to deal with. For a sufficiently large application the amount of C code is going to be comparable to the amount of C# code.

The main advantage of C# comes from the reduced number of files and the absence of complex linker options and header files.

Student Activity 2

1. What are the steps required in generating documentation? Explain briefly.
2. What are the best features that .NET components offer?

9.6 NGSW COMPONENTS

It is a new set of software, services and solutions for Microsoft customers, built around the Internet, Windows and new devices. Its design goals include support for unified platforms and tools effort; to build on the Windows 2000 generation of products; to create breakthrough software

and services. Next Generation Web Services (NGWS) will simplify the development of enterprise Web applications. Key to this strategy are Windows 2000 and new Web Services, Windows DNA 2000, ASP+ Web Forms and a number of language innovations. It is also known by its more formal name - Microsoft.NET.

Here we will develop a reusable windows service information component, which queries & retrieves the window's service information.

In .NET framework library the System.ServiceProcess namespace provides classes that allow you to implement, install, and control Windows service applications. To implement a service we need to inherit from ServiceBase class. In this article we are not going to implement a service; but we are going to develop a component which will retrieve & query the existing services. In order to proceed with this we will use ServiceController class (alias System.ServiceProcess.ServiceController). The ServiceController class enables you to connect to an existing service and manipulate it or get information about it.

So our component helps in administering the services running in the local system. The ServiceBase class defines the processing, a service performs when a command occurs. The ServiceController is the agent that enable us to call those commands on the services. So we are going to create a component which is derived from System.ComponentModel.Component, the default implementation of IComponent.

IComponent serves as the base class for all components in the common language runtime and IComponent allows a component to keep track of design-time information, such as its container component or its name, or to access services that the designer may expose.

Component class is remotable and derives from MarshalByRefObject which enables access to objects across application domain boundaries in applications that support remoting. Lets discuss first about ServiceInfo component. ServiceInfo Class is derived from Component class of .NET framework and this component will have read/write properties for getting/setting the Service information. So I have selected important methods of ServiceController class methods to query the services.

```
Namespace: ServiceInfoLib
Class:   ServiceInfo
```

Properties:

1. DisplayType (get/set property)

Methods:

1. GetNonDDServicesDetails() (gets the Non Device Driver Service Details)
2. GetDDServicesDetails() (gets the Device Driver Service Details)
3. GetNonDDServicesStatus() (gets the Non Device Driver Service Status)
4. GetDDServicesStatus() (gets the Device Driver Service Status)
5. FindService(string s) (Checks whether service exists or not)

Most of the methods in ServiceInfo component returns a string array. Internally all the methods of the component will call the ServiceController methods.

These members have been detailed below.

- The GetNonDDServiceDetails() method of ServiceInfo component returns a list of all the Non Device Driver Services installed on your local system. It internally calls the SystemController.GetServices() method. It also uses the property called DisplayType in ServerInfo component where the property get/set the display name of the service. If it is set to display type 'D' then it gives the list of friendly display Name of the Services and if its set to 'S' then it gives the list of Service Name of the Services; internally it calls the ServiceController DisplayName and ServiceName properties.

- The GetDDServiceDetails() method of ServiceInfo component returns a list of all the Device Driver Service installed on your local system. It internally calls the SystemController.GetDevices() method.
- The GetNonDDServiceStatus() method of ServiceInfo component returns a list of all the Non Device Driver Service installed on your local system and their status, which indicates whether the service is running, stopped, or paused, or whether a start, stop, pause, or continue command is pending. It internally calls the SystemController.GetServices() method and ServiceController.Status property which returns the ServiceControllerStatus class which indicates the status. In the component, I used ResolveSrvStatus method to resolve the exact status.
- The GetDDServiceStatus() method of ServiceInfo component returns a list of all the Device Driver Services installed on your local system and their status which indicates whether the service is running, stopped, or paused, or whether a start, stop, pause, or continue command is pending. It internally calls the SystemController.GetDevices() method and ServiceController.Status property which returns the ServiceControllerStatus class which indicates the status. In the component I used ResolveSrvStatus method to resolve the exact status.
- The FindService(string s) method of ServiceInfo component is to check whether a particular service exists in the local machine or not; if its found it returns a string saying 'Found' and if that particular service does not exist, then it returns 'Not Found'

The source code:

```
// SrvInfoLib.cs
// ServiceInfo component code in c#
using System;
using System.ComponentModel;
using System.ServiceProcess ;
namespace ServiceInfoLib
{
    /// <summary>
    /// ServiceInfo Class gets the windows service details.
    /// </summary>
public class ServiceInfo: Component
{

    private static char cDTyp='D'; // 'D' -> Display Name
    // 'S' -> Service Name

    public char DisplayType
    {
        set
        {
            cDTyp=value;
        }
        get
        {
            return cDTyp;
        }
    }
    public ServiceInfo(){}
    //Retrieves a list of Non Device Driver Services
    public  string[] GetNonDDServicesDetails()
    {

        string[] s=null;
        try
```

```
        {

            ServiceController[] srvC=
                ServiceController.GetServices();
            s=new string[srvC.Length];

            for(int i=0; i < srvC.Length; i++)
             {
                if(cDTyp=='D')
                    s[i]=srvC[i].DisplayName;
                else if(cDTyp=='S')
                    s[i]=srvC[i].ServiceName;
             }
        }
        catch(Exception x){}
        return s;
    }

    //Retrieves a list of  Device Driver Services
    public  string[] GetDDServicesDetails()
    {

        string[] s=null;
        try
        {

ServiceController[] srvC=ServiceController.GetDevices();
            s=new string[srvC.Length];
            for(int i=0; i < srvC.Length;i++)
             {

                if(cDTyp=='D')
                    s[i]=srvC[i].DisplayName;
                else if(cDTyp=='S')
                    s[i]=srvC[i].ServiceName;
             }
        }
        catch(Exception x){}
        return s;

    }

    //Retrieves a list of Non Device Driver Services
    //and status
    public  string[] GetNonDDServicesStatus()
    {
        string[] s=null;
        try
        {

           ServiceController[] srvC=ServiceController.GetServices();
            s=new string[srvC.Length];
            for(int i=0; i < srvC.Length; i++)
             {
                if(cDTyp=='D')
                    s[i]=srvC[i].DisplayName + "\t->" +
```

```
                    ResolveSrvStatus(srvC[i].Status);
                else if(cDTyp=='S')
                    s[i]=srvC[i].ServiceName + "\t->" +
                    ResolveSrvStatus(srvC[i].Status);
            }
        }
        catch(Exception x){}
        return s;

    }

    //Retrieves a list of  Device Driver Services
    //and  status
    public  string[] GetDDServicesStatus()
    {
        string[] s=null;
        try
        {

           ServiceController[] srvC=ServiceController.GetDevices();
            s=new string[srvC.Length];
            for(int i=0;i < srvC.Length; i++)
            {
                if(cDTyp=='D')
                    s[i]=srvC[i].DisplayName+"\t->" +
                    ResolveSrvStatus(srvC[i].Status);
                else if(cDTyp=='S')
                    s[i]=srvC[i].ServiceName+"\t->" +
                    ResolveSrvStatus(srvC[i].Status);
            }
        }
        catch(Exception x){}
        return s;
    }
    //Checks for particular service exists or not
    public string FindService(string s)
    {
        string s3=null;
        try
        {
            ServiceController[] services;
            services = ServiceController.GetServices();
            for(int i = 0; i <  services.Length; i++)
            {
                if(services[i].ServiceName==s)
                {
                    s3="Found";
                    break;
                }
            }
        }
        catch(Exception x){}
        if(s3==null)
            return s3="Not Found";
        else
            return s3;
```

```
    }
   private string ResolveSrvStatus(ServiceControllerStatus sl)
    {
        string st="";
       if(sl==ServiceControllerStatus.ContinuePending)
            st="The service continue is pending.";
       if(sl==ServiceControllerStatus.Paused)
            st="The service is paused.";
       if(sl==ServiceControllerStatus.PausePending)
            st="The service pause is pending.";
       if(sl==ServiceControllerStatus.Running)
            st="The service is running.";
       if(sl==ServiceControllerStatus.StartPending)
            st="The service is starting.";
       if(sl==ServiceControllerStatus.Stopped)
            st="The service is stopped.";
       if(sl==ServiceControllerStatus.StopPending)
            st="The service is not running.";
        return st;
    }

}

}
```

Once the above code is compiled as a component library to produce ServiceInfoLib.dll, one can call this component in different clients like Windows Forms, Web Forms or Console applications.

Let us create a simple Windows Forms application to use this component. To use the component in the client, add a reference in Visual Studio.NET and add using ServiceInfoLib; in the client code and create a instance of the component

```
( ServiceInfo srv = new ServiceInfo() )
```

and then use the methods of the component. Following is the code snippet of button click events of the client application

```
private void bNonDDSrvDet_Click(object sender, System.EventArgs e)
{
    tDisp.Clear(); //textbox

    //Invokes ServiceInfo component Non Device Driver details method
    string[] s=srv.GetNonDDServicesDetails();

    for (inti=0; i<s.Length; i++)
    {
        tDisp.Text+=(s[i]+"\r\n");
    }
}

private void bDDSrvDet_Click(object sender, System.EventArgs e)
{
    tDisp.Clear(); //textbox

    //Invokes ServiceInfo component Device Driver details method
    string[] s=srv.GetDDServicesDetails();
    for(int i=0;i<s.Length;i++)
    {
        tDisp.Text+=(s[i]+"\r\n");
```

```
    }
}

private void bNonDDSrvStat_Click(object sender, System.EventArgs e)
{
   tDisp.Clear();//textbox
   //Invokes ServiceInfo component Non Device Driver status method
   string[] s=srv.GetNonDDServicesStatus();
   for(int i=0;i<s.Length;i++)
    {
       tDisp.Text+=(s[i]+"\r\n");
    }
}

private void bDDSrvStat_Click(object sender, System.EventArgs e)
{
    tDisp.Clear();    //textbox

   //Invokes ServiceInfo component Device Driver status method
   string[] s=  srv.GetDDServicesStatus();
   for(int i=0;i<s.Length;i++)
    {
       tDisp.Text+=(s[i]+"\r\n");
    }
}

private void bDDSrvFind_Click(object sender, System.EventArgs e)
{
   tDisp.Clear(); //textbox

    //check for Alerter service exists or not
   tDisp.Text=srv.FindService("Alerter");
}
//setting the Display type property of ServiceInfo component
private void rbDispName_CheckedChanged(object sender, System.EventArgs
e)
{
   srv.DisplayType='D'; //friendly Display Name
}

//setting the Display type property of ServiceInfo component
private void rbSrvName_CheckedChanged(object sender, System.EventArgs
e)
{
   srv.DisplayType='S'; // Actual instance Service Name
}
```

You can now start, pause and stop this service on a machine in the same way any Windows service is operated on.

9.7 SUMMARY

- Preprocessing directives are lines in your program that start with `#'. Whitespace is allowed before and after the `#'. The `#' is followed by an identifier that is the directive name.
- #define lets you define a symbol, such that, by using the symbol as the expression passed to the #if directive, the expression will evaluate to true

- #undef lets you undefine a symbol, such that, by using the symbol as the expression in a #if directive, the expression will evaluate to false.
- .NET is a great improvement in COM technology as it brings us nearer to true code reusability. But there is a great need for interoperability between the COM and .NET components.
- NGSW components are a new set of software, services and solutions for Microsoft customers, built around the Internet, Windows.

9.8 KEYWORDS

- **Preprocessor:** It is a utility that processes the source code file(s) before the actual compilation begins. A number of actions are taken by the preprocessor to produce the output in the desirable format of the compiler.
- **Documentation:** Manuals, tutorials, and Help files that provide information that a user needs in order to use a computer system or software application
- **COM:** Component Object Model (COM) is Microsoft's object-oriented programming model that defines how objects interact within a single application or between applications
- **NGSW components:** It is a new set of software, services and solutions for Microsoft customers, built around the Internet, Windows.

9.9 REVIEW QUESTIONS

1. Explain the directives
   a) `#error' and `#warning' D
   b) #line
   c) #define directive
   d) #undef directive
2. What are some of the tools for interoperability between .Net and COM?
3. What are NSGW components?
4. Name the methods of the ServiceInfo component. Explain briefly the methods.

9.10 FURTHER READINGS

CLR via C#, Second Edition (Paperback) by Jeffrey Richter, Publisher: Microsoft Press; 2nd edition (February 22, 2006)

Essential C# 2.0 (Microsoft .NET Development Series) (Paperback) by Mark Michaelis, Publisher: Addison-Wesley Professional (July 13, 2006)

C# Programming Language, The (2nd Edition) (Microsoft .NET Development Series) (Hardcover) by Anders Hejlsberg, Scott Wiltamuth, Peter Golde, Publisher: Addison-Wesley Professional; 2 edition (June 9, 2006)

Programming in the Key of C# (Paperback) by Charles Petzold, Publisher: Microsoft Press; 1 edition (July 30, 2003)

UNIT

10

SECURITY

L E A R N I N G O B J E C T I V E S

After studying this unit, you should be able to

- Define security
- Verify type security
- Understand code access security
- Define role based security and tools

U N I T S T R U C T U R E

10.1 INTRODUCTION

This unit will introduce you to .NET Framework rich security system which is capable of confining code to run in tightly constrained administrator-defined security contexts. You will learn about some of the fundamental security features provided in the .NET Framework.

10.2 SECURITY

Security is one of the most important issues to consider when moving from traditional program development, where administrators often install software to a fixed location on a local disk, to an environment that allows for dynamic downloads and execution and even remote execution. To support this model, the Microsoft .NET Framework provides a rich security system, capable of confining code to run in tightly constrained, administrator-defined security contexts. Here we will examine some of the fundamental security features in the .NET Framework.

Many security models attach security to users and their groups (or roles). This means that users, and all code run on behalf of these users, are either permitted or not permitted to perform operations on critical resources. This is how security is modeled in most operating systems. The .NET Framework provides a developer defined security model called role-based security that functions in a similar way. Role Based Security's principal abstractions are Principals and Identity. Additionally, the .NET Framework also provides security on code and this is referred to as code access security (also referred to as evidence-based security). With code access security, a user may be trusted to access a resource but if the code the user executes is not trusted, then access to the resource will

be denied. Security based on code, as opposed to specific users, is a fundamental facility to permit security to be expressed on mobile code. Mobile code may be downloaded and executed by any number of users all of which are unknown at development time. Code Access Security centers on some core abstractions, namely: evidence, policies, and permissions. The security abstractions for role-based security and code access security are represented as types in the .NET Framework Class Library and are user-extendable. There are two other interesting challenges: to expose the security model to a number of programming languages in a consistent and coherent manner and to protect and expose the types in the .NET Framework Class Library that represent resources whose use can lead to security breaches.

The .NET Framework security system functions atop traditional operating system security. This adds a second more expressive and extensible level to operating system security. Both layers are complementing each other. (It is conceivable that an operating system security system can delegate some responsibility to the common language runtime security system for managed code, as the runtime security system is finer grain and more configurable then traditional operating system security.) The .NET Framework security system is highly configurable and extensible. This is a major strength of the system.

Overview of Execution

The runtime executes both managed and unmanaged code. Managed code executes under the control of the runtime and therefore has access to services provided by the runtime, such as memory management, just-in-time (JIT) compilation, and, most importantly security services, such as the security policy system and verification.

Unmanaged code is code that has been compiled to run on a specific hardware platform and cannot directly utilize the runtime. However, when language compilers emit managed code, the compiler output is represented as Microsoft intermediate language (MSIL). MSIL is often described as resembling an object-oriented assembly language for an abstract, stack-based machine. MSIL is said to be object-oriented, as it has instructions for supporting object-oriented concepts, such as the allocation of objects (newobj) and virtual function calls (callvirt). It is an abstract machine, as MSIL is not tied to any specific platform. That is, it makes no assumptions about the hardware on which it runs. It is stack-based, as essentially MSIL executes by pushing and popping values off a stack and calling methods. MSIL is typically JIT-compiled to native code prior to execution. MSIL can also be compiled to native code prior to running that code. This can help start-up time of the assembly, though typically MSIL code is JIT-compiled at the method level.

10.3 VERIFICATION OF TYPE SECURITY

Type-safe code accesses only the memory locations it is authorized to access. It cannot, for example, read values from another object's private fields. Type-safe code accesses types only in well-defined, allowable ways. During Just-In-Time (JIT) compilation, an optional verification process examines the Microsoft Intermediate Language (MSIL) in an attempt to verify that the MSIL is type-safe. This process is skipped if the code has permission to bypass verification.

Code that is not verifiably type-safe can attempt to execute if security policy allows the code to bypass verification. However, because type-safety is an essential part of the runtime's mechanism for isolating assemblies, executing unverifiable code can cause problems that crash other applications as well as the runtime itself. Also, security cannot be reliably enforced if the code violates the rules of type-safety.

There are two forms of verification done in the runtime. MSIL is verified and assembly metadata is validated. All types in the runtime specify the contracts that they will implement, and this information is persisted as metadata along with the MSIL in the managed PE/COEFF file.

For example, when a type specifies that it inherits from another class or interface, indicating that it will implement a number of methods, this is a contract. A contract can also be related to visibility. For example, types may be declared as public (exported) from their assembly or not. Type safety is a property of code in so much as it only accesses types in accordance with their

contracts. MSIL can be verified to prove it is type safe. Verification is a fundamental building block in the .NET Framework security system; currently verification is only performed on managed code. Unmanaged code executed by the runtime must be fully trusted, as it cannot be verified by the runtime.

In order to understand MSIL verification, it is important to understand how MSIL can be classified. MSIL can be classified as invalid, valid, type safe, and verifiable.

Invalid MSIL is MSIL for which the JIT compiler cannot produce a native representation. For example, MSIL containing an invalid opcode could not be translated into native code. Another example would be a jump instruction whose target was the address of an operand rather than an opcode.

Valid MSIL could be considered as all MSIL that satisfies the MSIL grammar and therefore can be represented in native code. This classification does include MSIL that uses non-type-safe forms of pointer arithmetic to gain access to members of a type.

Type-safe MSIL only interacts with types through their publicly exposed contracts. MSIL that attempted to access a private member of a type from another type is not type-safe.

Verifiable MSIL is type-safe MSIL that can be proved to be type-safe by a verification algorithm. The verification algorithm is conservative, so some type-safe MSIL might not pass verification. Naturally, verifiable MSIL is also type-safe and valid but not, of course, invalid.

In addition to type-safety checks, the MSIL verification algorithm in the runtime also checks for the occurrence of a stack underflow/overflow, correct use of the exception handling facilities, and object initialization.

For code loaded from disk, the verification process is part of the JIT compiler and proceeds intermittently within the JIT compiler. Verification and JIT compilation are not executed as two separate processes. If, during verification, a sequence of unverifiable MSIL is found within an assembly, the security system checks to see if the assembly is trusted enough to skip verification. For example, if an assembly is loaded from a local hard disk, under the default settings of the security model, then this could be the case. If the assembly is trusted to skip verification, then the MSIL is translated into native code. If the assembly is not trusted enough to skip verification then the offending MSIL is replaced with a stub that throws an exception if that execution path is exercised. A commonly asked question is, "Why not check if an assembly needs to be verified before verifying it?" In general, verification, as it is done as part of the JIT compilation, is often faster than checking to see if the assembly is allowed to skip verification. (The decision to skip verification is smarter than the process described here. For example, results of previous verification attempts can be cached to provide a quick lookup scenario.)

In addition to MSIL verification, assembly metadata is also verified. In fact, type safety relies on these metadata checks, for it assumes that the metadata tokens used during MSIL verification are correct. Assembly metadata is either verified when an assembly is loaded into the Global Assembly Cache (GAC), or Download Cache, or when it is read from disk if it is not inserted into the GAC. (The GAC is a central storage for assemblies that are used by a number of programs. The download cache holds assemblies downloaded from other locations, such as the Internet.) Metadata verification involves examining metadata tokens to see that they index correctly into the tables they access and that indexes into string tables do not point at strings that are longer than the size of buffers that should hold them, eliminating buffer overflow. The elimination, through MSIL and metadata verification, of type-safe code that is not type-safe is the first part of security on the runtime.

10.4 PERMISSIONS

The common language runtime (CLR) allows code to perform only those operations that the code has permission to perform. CLR enforces restrictions on managed code through permission objects. Code can demand that its callers have specific permissions. If we place a demand for certain permission on our code, all codes that use our code must have that permission to run.

Code can request the permissions it needs for the execution. The runtime grants permission to code based on characteristics of the code's identity and on how much the code is trusted. The level of Trust is determined by security policy set by an administrator.

.NET provides the following kinds of permissions (called standard permissions):

1. ***Code access permissions:*** These permissions represent access to a protected resource (e.g. HDD) or the ability to perform a protected operation. Some of the code access permissions are:

WebPermission	The ability to make/accept connection to/from the web.
OleDbPermission	The ability to access databases with OLEDB.
SQLClientPermission	The ability to access SQL databases.
UIPermission	The ability to use User Interface.
PrintingPermission	The ability to print.
FileIOPermission	The ability to work with files.
SocketPermission	The ability to make/accept TCP/IP connections on a transport address.
SecurityPermission	The ability to execute, assert permissions, call into unmanaged code, skip verification and other rights.

2. ***Permission Sets:*** These are the lists of code access permissions grouped into a named set.

Internet	Default policy for code of unknown origin
LocalIntranet	Default policy for the local intranet
FullTrust	No restriction on the permission
Everything	All standard permission except the permission to skip code verification
Execution	Permission to execute but no access to any protected resources
Nothing	No Permission - unable to execute

3. ***Identity permissions:*** These permissions indicate that code has credentials that support a particular kind of user identity. Some of the identity permissions provided by CLR are listed below.

PublisherIdentityPermission	Software publisher's digital signature
StrongNameIdentityPermission	Assembly's strong Name
URLIdentityPermission	URL from which the code is originated
ZoneIdentityPermission	Zone from which the assembly is originated
SiteIdentityPermission	Location of web site from which the code is originated

4. ***Role-based security permissions:*** These permissions provide a mechanism for discovering whether a user has a particular identity or is a member of a specified role. PrincipalPermission is the only role-based security permission.

Let take a sample application, which attempts to access a disk file and an environment variable. Code shown below will create permission to set read access to Temp environment and full access to some files.

```
 //Create a permission set that allows read access to the TEMP
//environment variable and read, write, and append access to SomeFile from
//default permission
  PermissionSet ps = new PermissionSet(PermissionState.None);
  ps.AddPermission(
    new EnvironmentPermission(EnvironmentPermissionAccess.Read, "TEMP"));
//adding various type of file level permission
  ps.AddPermission(
    new FileIOPermission(FileIOPermissionAccess.Read |
    FileIOPermissionAccess.Write | FileIOPermissionAccess.Append,
    "SomeFile"));
// Make the permissions indicate all that we're allowed to do.
        ps.Assert();
```

PermissionSet class (in System.security) represents a collection and it contains many different kinds of permissions, and supports the methods that use and modify those permissions. We can add, remove, assert, deny and copy permission.

```
//Deny access to the resources we specify
        ps.Deny();
//Make the permissions indicate the only things that we're allowed to do.
        ps.PermitOnly();
//Remove the FileIOPermissions from the permission set
       ps.RemovePermission(typeof(FileIOPermission))
//Remove the EnvironmentPermission from the permission set
       ps.RemovePermission(typeof(EnvironmentPermission));
```

Deny method prevents callers from accessing the protected resource even if they have been granted permission to access it. PemitOnly Ensures that only the resources specified by this permission object can be accessed, even if the code has been granted permission to access other resources. FileIOPermissionAccess specifies the actions that can be performed on the file or folder. EnvironmentPermission Class as the ability to query and modify system and user environment variables.

A heirarchy of permissions and their relationship is depicted below.

1. Effective permission is the intersection of permissions from different levels. If we want to apply FullTrust to code group we will have to assign this permission on each of the levels.
2. Each of the three levels has the ability to ban the permission allowed by the another
3. By default, user level and enterprise level are configured to allow FullTrust for the single code group All Code. Machine level policy is the default policy. These settings place no restrictions at the enterprise or user level.

Student Activity 1

1. What are the forms of verification done in the runtime?
2. How are MSIL classified? Explain verifiable MSIL.
3. What does metadata verification involve? When is the assembly metadata verified?
4. What does the MSIL verification algorithm do?

10.5 CODE ACCESS SECURITY

Code access security helps protect computer systems from malicious mobile code, to allow code from unknown origins to run safely and to protect trusted code. Code access security enforces varying levels of trust on the code. It reduces the likelihood that our code can be misused by other malicious code. With the help of code access security, we can specify the set of operations that our code should be allowed to perform or should never be allowed to perform.

Code access security benefits all managed code that targets the CLR. Code access security performs the following functions:

- It defines permissions and permission sets to access various system resources.
- It enables administrators to configure security policy.
- It allows code to request the permissions.
- It grants permissions to each assembly.
- It enables code to demand that its callers have specific permissions.
- It enables code to demand that its callers possess a digital signature.

Security stack walk

The above figure illustrates the stack walk that results when a method in Assembly A4 demands that its callers have permission P. The runtime's security system walks the call stack to determine whether code is authorized to access a resource or perform an operation, It compares the granted permissions of each caller to the permission being demanded. If any caller in the call stack does

not have the permission that was demanded, a security exception is thrown and access is refused. Demanding permissions of all callers at run time affects performance, but is essential to protect code from luring attacks by less trusted code.

The local security settings on a particular machine ultimately decide which permissions code receives. These settings can change from machine to machine, so, we can never be sure that your code will receive sufficient permissions to run. In case of unmanaged code, developers don't have to worry about their code's permission to run.

Code groups

Code groups bring together code that has similar characteristics. An assembly is associated with several code groups. However, for an assembly to be filled in a code group, it must match the group's membership condition. Each code group can have one and only one membership condition. Code groups are arranged in hierarchy with All Code membership condition at the root. The membership conditions are listed below.

Zone	Region (Internet, Intranet, trusted and untrusted) from which code is originated
Site	Web site from which code is originated
Strong name	Unique, verifiable code name
Publisher	Publisher
URL	URL from which the code originated
Skip verification	Code that request to bypass verification check
Application directory	Assembly location
All code	All code fulfills this condition
Custom	User-Defined condition.

Zones are managed using Internet Explorer. (Tools à Internet Options à Security) They apply to the entire machine. These options are not available in non-Microsoft browsers. In-page controls written using .NET Framework does not work in browsers other than Internet Explorer. Code groups are listed on three levels - Enterprise, Machine and User.

Code Access Security Policy Tool (caspol.exe)

Caspol.exe lets us view and manage security policy. Entering "caspol" on the command gives us a list all the options available with caspol.exe.

Caspol.exe -listdescription	Lists the hierarchical structure of the code groups on the machine
Caspol.exe -listgroups	Gives a compact view of code access groups
Caspol.exe -resolvegroup assembly.dll	Gives a list of code groups that an assembly is a member of
Caspol.exe -resolveperm assembly.dll	Give's the permissions of an assembly's code groups.
Caspol.exe --user -listgroups	Gives the code groups listing at the user level
Caspol.exe --Enterprise -listgroups	Gives the code groups listing at the Enterprise level
Caspol.exe --security off	Disables the .NET security
Caspol.exe --security off	Enables the .NET security

Creating a code group

Suppose, we want to trust all codes from www.microsodt.com and give it full access to our system. It's a two-step process

1. Get the numeric label of the code group within which our new code group will live. This can be done as caspol.exe -listgroups. Say, using the fact zone, Internet is labeled as 1.3
2. casplo.exe -addgroup 1.3 - site www.microsoft.com adds the code group.

Deleting a code group

```
casplo.exe -remgroup 1.3.2 will remove the code group we have just created,
```

Changing code group's permissions

We can also change permissions of a particular group. It's a two-step process

1. Get the numeric label of the required code group using caspol.exe -listgroups. Say, zone Intranet has LocalIntranet permissions and has been labeled as 1.2 then,
2. casplo.exe -chggroup 1.2 - FullTrust will change the permissions

Security Syntax

.NET Framework has provided declarative syntax and imperative syntax to programmatically interact with the security system. Code that targets the common language runtime can interact with the security system by Demanding resources, Requesting resources and Overriding certain security settings. This section also covers denying permission, asserting permission and implicit permission.

Demanding Permissions

Demands are applied on the class and method level as a mechanism to ensure that callers of our code have the permissions that we want them to have. When our code is called, demands invoke a stack walk, in which all callers that directly or indirectly call our code are checked on the stack. Demands are usually used in class libraries to protect resources (like local disk, etc.).

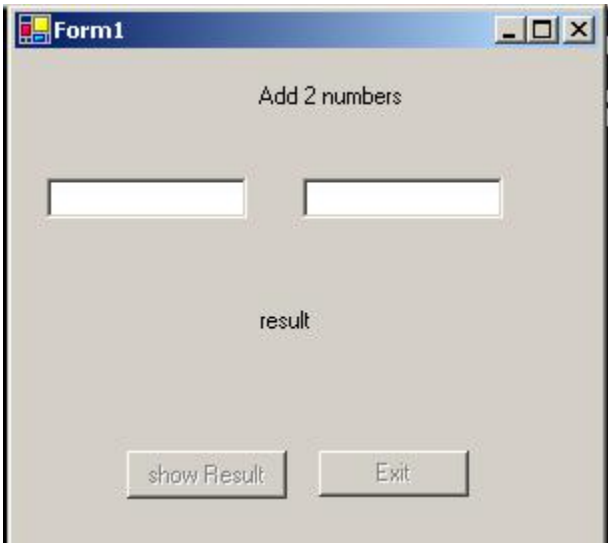
Simply, demanding permissions means stating clearly in the program what we need at runtime.

We can take any one of our earlier programs and just by adding a little code we can impose security constraints on our program. Let us take our earlier program that adds two numbers. While creating an object of FIleIOPermission class we can specify whether we need full access or read only access.

Now if we run the application from local disk, where the default security policy allows access to local storage, the 'result' and 'exit' button will be enabled. However, if we try to run this application from a network share we are operating within a LocalIntranet Permission sets, which blocks access to local storage and the button will be grayed.

Example

Runtime image of the Form while running from the Intranet




```
using System;
using System.Drawing;
using System.Collections;
```

```
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.Security;
using System.Security.Permissions;
namespace WindowsApplication1
{
public class Form1 : System.Windows.Forms.Form
{ private System.Windows.Forms.Label label1;
private System.Windows.Forms.Button btnResult;
private System.Windows.Forms.Button btnExit;
private System.Windows.Forms.TextBox txtNum1;
private System.Windows.Forms.TextBox txtNum2;
private System.Windows.Forms.Label lblResult;
private EventHandler handler;
private System.ComponentModel.Container components = null;
public Form1()
{ InitializeComponent();
handler = new EventHandler(oneFunction);
// additional CODE required for Security options STARTS here
try
{
FileIOPermission myPermission = new FileIOPermission
                          (FileIOPermissionAccess.AllAccess,@"c:\\");

myPermission.Demand();
}
catch
{
btnResult.Enabled = false;
btnExit.Enabled = false;
}
// additional CODE required for Security options ENDS here
}
protected override void Dispose( bool disposing )
{
if( disposing )
{
if (components != null)
{ components.Dispose();     }
}
base.Dispose( disposing );
}
#region Windows Form Designer generated code
private void InitializeComponent()
{ this.label1 = new System.Windows.Forms.Label();
this.btnResult = new System.Windows.Forms.Button();
this.btnExit = new System.Windows.Forms.Button();
this.txtNum1 = new System.Windows.Forms.TextBox();
this.txtNum2 = new System.Windows.Forms.TextBox();
this.lblResult = new System.Windows.Forms.Label();
this.SuspendLayout();
this.label1.Location = new System.Drawing.Point(120, 16);
this.label1.Name = "label1";
this.label1.TabIndex = 0;
this.label1.Text = "Add 2 numbers";
```

```
this.btnResult.Location = new System.Drawing.Point(56, 200);
this.btnResult.Name = "btnResult";
this.btnResult.Size = new System.Drawing.Size(80, 24);
this.btnResult.TabIndex = 1;
this.btnResult.Text = "show Result";
this.btnResult.Click += new System.EventHandler(this.btnResult_Click);

this.btnExit.Location = new System.Drawing.Point(152, 200);
this.btnExit.Name = "btnExit";
this.btnExit.TabIndex = 2;
this.btnExit.Text = "Exit";
this.btnExit.Click += new System.EventHandler(this.btnExit_Click);
this.txtNum1.Location = new System.Drawing.Point(16, 64);
this.txtNum1.Name = "txtNum1";
this.txtNum1.TabIndex = 3;
this.txtNum1.Text = "";
this.txtNum2.Location = new System.Drawing.Point(144, 64);
this.txtNum2.Name = "txtNum2";
this.txtNum2.TabIndex = 4;
this.txtNum2.Text = "";
this.lblResult.Location = new System.Drawing.Point(120, 128);
this.lblResult.Name = "lblResult";
this.lblResult.Size = new System.Drawing.Size(80, 23);
this.lblResult.TabIndex = 5;
this.lblResult.Text = "result";
this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
this.ClientSize = new System.Drawing.Size(292, 273);
this.Controls.AddRange(new System.Windows.Forms.Control[] {
  this.lblResult,
this.txtNum2,
this.txtNum1,
this.btnExit,
this.btnResult,
this.label1});
this.Name = "Form1";
this.Text = "Form1";
this.Load += new System.EventHandler(this.Form1_Load);
this.ResumeLayout(false);
}
#endregion
static void Main()
{
Application.Run(new Form1());
}
private void btnResult_Click(object sender, System.EventArgs e)
{
btnResult.Click += handler;
}
private void btnExit_Click(object sender, System.EventArgs e)
{
btnExit.Click += handler;
}
private void oneFunction(Object source, EventArgs e)
{
if(source==btnExit)
Application.Exit();
```

```
if(source==btnResult)
{
try
{
lblResult.Text=" " +(Int32.Parse(txtNum1.Text)+Int32.Parse(txtNum2.Text));

}
catch
{  MessageBox.Show("please enter correct values...!!");  }
}}}}
```

Requesting Permissions

While demanding permissions we state quite clearly what we need at the RUNTIME. However, we can configure an assembly so that it makes a softer request right at the start of execution where it stats what it needs before it begins executing.

We request permissions for an assembly by placing attributes on the assembly scope of our code. We can request permissions in three ways - Minimum, Refused and Optional permissions. In the above example we can add the following lines of code:

```
//"Minimum permission".If it is not granted the application will fail to
start

[assembly : UIPermissionAttribute (SecurityAction. RequestMinimum)]

//"Refused permission" The entire assembly will be blocked from accessing
this drive.

[assembly : FileIOPermissionAttribute (SecurityAction.
RequestRefuse,Read="C:\\")]

//"Optional permission" If it is not granted the application will still
run

[assembly : SecurityPermissionAttribute (SecurityAction.
RequestOptional,Flags=SecurityPermissionFlag.UnmanagedCode)]
```

A request only influences the runtime to DENY permissions to our code and never influences the runtime to GIVE more permission to our code. The local administration policy always has final control over the maximum permissions our code is granted.

Overriding security settings

Security overrides are applied on the class and method scope to overrule certain security decisions made by the runtime. They are invoked when callers use our code. They are used to stop stack walks and limit the access of callers who have already been granted certain permissions. Overrides could be dangerous and should be used with care.

We can use a combination of security demands and security overrides in order to improve the performance during the interaction of our code and the security system. This can be referred to as 'Security Optimizations'

Denying Permission

When we want to make a call to third party class, denying permissions help us to make sure that the method we have called is acting within a protected environment.

In the following example we are reading a text file from the c:\ drive. We can see that we cannot open the text file when we deny the permission to access c:\. The sample output looks like this:

```
c:\>example
before deny
```

```
File opened : this is a text file with name - README.txt
after deny
FAILED TO OPEN FILE
This line is printed by another harmless method that does not access
C:|....
The code is listed as below:
using System;
using System.IO;
using System.Security;
using System.Security.Permissions;
namespace WindowsApplication3
{
public class myClass1
{
static void Main(string[] args)
{
Console.WriteLine("before deny");
myClass2.myOpenFile();
Now we will protect C drive. We can also protect a particular folder like
c:\Akash.
CodeAccessPermission myPermission = new
FileI OPermission(FileIOPermissionAccess.AllAccess,@"c:\");
Console.WriteLine("after deny");
myPermission.Deny();
```

This is a call on the instance of the permission object. Permission is denied to access C: drive. But, the user can still use other methods that do not access our resource.

```
myClass2.myOpenFile();
myClass2.print();}
CodeAccessPermission.RevertDeny();
}
}
public class myClass2
{
public static void myOpenFile()
{
try
{  StreamReader din = File.OpenText(@"C:\ReadMe.txt");
Console.WriteLine("FIle opened : "+din.ReadLine());
}
catch
{
Console.WriteLine("FAILED TO OPEN FILE");
}
}
public  static void  print()
{
Console.WriteLine("This line is printed by another harmless method that
does not access C:\\....");
}}}
```

Asserting Permission

Once the permission is denied, we can assert the permission temporarily. This can be done as follow. We have created an object 'myPermission' of FileIOPermission class. The methods Assert() and revertAssert() are used to give permission to append data to a file stored on the local disk temporarily.

Example

```
FileIOPermission myPermission = new
FileIOPermission(FileIOPermissionAccess.Append,@"c:\\ReadMe.txt");
myPermission.Assert();
FileStream myStrm = new FileStream(@"ReadMe.txt",FileMode.Append,
FileAccess.Write);
CodeAccessPermission.RevertAssert();
```

Implicit Permission

When permissions are granted to our code, some additional permission are also implicitly granted to us. When we are granted permission to access c:\ this means that we can access subdirectories of the c:\ like c:\Vijay.

Student Activity 2

1. What are the standard permissions provided by .NET?
2. How doe code access security help protect computer systems? What are the functions performed by the code access security?
3. What are code groups? How would you go about changing a code group's permission?
4. What is meant by security optimizations?

10.6 ROLE BASED SECURITY

Code access security gives the CLR the ability to make the decisions. In role-based security, the code can perform actions based on evidence about the user and his role. Role based security is especially useful in situations where the access to resources is an important issue. Role based security is ideal for use in conjunction with windows 2000 accounts.

Principal

The principal is at the core of the role-based security. It represents the identity and role of a user. Through Principal, we can access the user Identity from 1) user account types as windows account 2) passport and 3) ASP.NET cookie authenticated user. A Role is a collection of users who have same security permissions. Role is a unit of administration for user.

Role-based security in the .NET Framework supports three kinds of principals:

1. ***Windows principals:*** represent Windows users and their roles.
2. ***Generic principals:*** represent users and roles that exist independent of Windows NT and Windows 2000 users and roles.
3. ***Custom principals:*** can be defined by an application that is needed for that particular application. They can extend the basic notion of the principal's identity and roles.

All Identity classes implement the IIdentity interface. The IIdentity interface defines properties for accessing a name and an authentication type, such as Kerberos V5 or NTLM.

All Principal classes implement the IPrincipal interface as well as any additional properties and methods that are necessary

The principal object represents the security context under which code is running. Applications that implement role-based security grant rights based on the role associated with a principal object.

Authentication

Authentication is the process of discovering and verifying the identity of a principal. Once the identity of the principal is discovered, we can use role-based security to determine whether to allow that principal to access our code. Digest, Passport, operating system (such as NTLM or Kerberos), or application-defined mechanisms are some of the commonly used authentication mechanisms.

Authorization

Authorization follows authentication. It is the process of determining whether a principal is allowed to perform a requested action. Information about the principal's identity and roles is used to determine what resources the principal can access. We can use .NET Framework role-based security to implement an authorization scheme.

Creating GenericPrincipal and GenericIdentity classes and objects

We can use the GenericIdentity class in conjunction with the GenericPrincipal class to create an authorization scheme that exists independent of a Windows NT or Windows 2000 domain. For example, an application that uses these two objects might prompt a user for a name and password, check them against a database entry, and create identity and principal objects based on the values in the database.

A GenericIdentity object can be used for most custom logon scenarios. We can define our own identity class that encapsulates custom user information.

Example

In this example we are creating generic identity and generic principal and apply them to our current thread. The results are displayed on console. This code can be used to create an authorization scheme for a particular application.

The output of the program is given below.

```
C:\>Example
The Identity is: ArunIdentity
The IsAuthenticated is: True
Is this a manager? True
The code for the same is:
using System;
using System.Security.Principal;
using System.Threading;
public class Class1
{
     public static int Main(string[] args)
      {
//Create instances of generic identity and generic principal class.
GenericIdentity myIdentity = new GenericIdentity("ArunIdentity");
String[] myStringArray = {"Manager", "Professor"};
GenericPrincipal  myPrincipal  =  new  GenericPrincipal(myIdentity,
myStringArray);
//Attach the principal to the current thread.
Thread.CurrentPrincipal = myPrincipal;
      String strName =  myPrincipal.Identity.Name;
      bool blnAuth =  myPrincipal.Identity.IsAuthenticated;
      bool blnIsInRole =  myPrincipal.IsInRole("Manager");
      Console.WriteLine("The Identity is: {0}", strName);
      Console.WriteLine("The IsAuthenticated is: {0}", blnAuth);
      Console.WriteLine("Is this a Manager? {0}", blnIsInRole);
       return 0;
       }}
```

10.7 .NET SECURITY TOOLS

.NET Framework has the following security tools. To run these exe files we first need to set the path (llike C:\Program Files\Microsoft.NET\FrameworkSDK\Bin).

Tool	Description
Code Access Security Policy Tool (Caspol.exe)	Enables to view and configure security policy.
Certificate Creation Tool (Makecert.exe)	Generates X.509 certificates for testing purposes only.
Certificate Manager Tool (Certmgr.exe)	Manages certificates, certificate trust lists (CTLs), and certificate revocation lists (CRLs).
Certificate Verification Tool (Chktrust.exe)	Checks the validity of a file signed with an Authenticode™ certificate.
Permissions View Tool (Permview.exe)	Allows us to view an assembly's requeste permissions.
PEVerify Tool (Peverify.exe)	Determines whether the JIT compilation process can verify the type safety of the assembly.
Set Registry Tool (Setreg.exe)	Changes the registry settings that pertain to certificates and digital signatures.
Software Publisher Certificate Test Tool (Cert2spc.exe)	Creates a Software Publisher's Certificate (SPC) from one or more X.509 certificates. This tool is for testing purposes only.
Strong Name Tool (Sn.exe)	Helps create assemblies with strong names. It provides options for key management, signature generation and verification

These tools are command-line utilities that help perform security-related tasks and test your components and applications before we deploy them.

Assembly Generation Utility (al.exe)

Al.exe takes as its input one or more files that are either in MSIL format or are resource files and outputs a file with an assembly manifest.

Code Access Security Policy Utility (caspol.exe)

The caspol utility enables users and administrators to modify security policy for the user policy level and the machine policy level.

Software Publisher Certificate Test Utility (Cert2spc.exe)

The Software Publisher Certificate test utility (Cert2spc.exe) creates a Software Publisher's Certificate (SPC) from one or more X.509 certificates. Note that Cert2spc, like Makecert, is for test purposes only. A valid SPC is obtained from a Certification Authority (CA) such as VeriSign or Thawte.

Certificate Manager Utility (certmgr.exe)

The Certificate Manager utility manages certificates, certificate trust lists (CTLs), and certificate revocation lists (CRLs). Certmgr performs the following basic functions:

- Displays certificates, CTLs, and CRLs to the console.
- Adds certificates, CTLs, and CRLs, from one certificate store to another.
- Deletes certificates, CTLs, and CRLs, from a certificate store.
- Saves an X.509 certificate, CTL, and CRL from a certificate store to a file.

Certificate Verification Utility (chktrust.exe)

The ChkTrust utility checks the validity of an Authenticode™-signed file. If the hashes agree, ChkTrust verifies the signing certificate.

Certificate Creation Utility (makecert.exe)

Makecert (the X.509 Certificate Creation utility) generates a X.509 certificate, which can be used for testing purposes only. It creates a public and private key pair for digital signatures and associates it with a name that you specify.

Permissions View Utility (permview.exe)

Use the PermView command line utility to view the minimal, optional, and refused permission sets requested by an assembly. By default, permission requests are dumped to the console.

PEVerify Utility (peverify.exe)

The PEVerify utility is provided to assist developers generating MSIL (i.e., compiler writers, script engine developers, etc.) in determining whether their MSIL code and associated metadata meets type-safety verification requirements.

Secutil Utility (SecUtil.exe)

The Microsoft .NET Common Language Runtime security system provides mechanisms for restricting the actions of code based on its associated evidence. Two types of evidence, strong names (also called shared names) and Authenticode™ publishers, are based on cryptographic keys and digital signature technology. As a result, we have high assurance concerning this evidence for a given assembly.

Set Registry Utility (setreg.exe)

Setreg (the Set Registry utility) changes registry settings for public key cryptography.

File Signing Utility (signcode.exe)

Signcode (the File Signing utility) signs a portable executable (PE) file with requested permissions to give developers more detailed control over the security restrictions placed on their component. You can sign a component or an assembly. If you are distributing an assembly rather than individual components (i.e., .dlls or .exes), you should sign the assembly, not the individual components. If signcode is run without any options, it launches a wizard to help with signing.

Isolated Storage Utility (storeadm.exe)

Use the StoreAdm command line tool to manage isolated storage The tool provides three simple functions that are typically used one at a time namely /LIST, /REMOVE and /QUIET.

Student Activity 3

1. What is role-based security permission? Where is role-based security most useful?
2. Explain what do you understand by "principal" in role-based security. What are the three kinds of principals supported by the .NET Framework?
3. What is meant by authentication? What are some of the commonly used authentication mechanism?
4. What is meant by Authorization? How can you use .NET Framework role-based security to implement an authorization scheme?
5. What are the .Net security tools?

10.8 SUMMARY

- The .NET Framework offers code access security and role-based security to help address security concerns about mobile code and to provide support that enables components to make decisions about what users are authorized to do
- There are three kinds of permissions: Code permissions, Identity Permissions & Role-based permissions

- Code that is not verifiably type-safe can attempt to execute if security policy allows the code to bypass verification
- A principal is a user or an agent that acts on the user's behalf.
- .NET Framework role- based security has support for three kinds of principals:

  Generic principals represent unauthenticated users and the roles available to them.
- Security Tools provided by .NET for its applications are command-line utilities that help perform security-related tasks and test your components and applications before we deploy them

10.9 KEYWORDS

- **Type-safety:** Type-safe code accesses only the memory locations it is authorized to access. It cannot, for example, read values from another object's private fields.
- **Authentication:** It is the process of discovering and verifying the identity of a principal by examining the user's credentials and validating those credentials against some authority
- **Authorization:** It is the process of determining whether a principal is allowed to perform a requested action.
- **.NET security tools:** Security Tools provided by .NET for its applications are command-line utilities that help perform security-related tasks and test your components and applications before we deploy them
- **Role Based Security:** It involves granting permissions based on a user's role(s) and groups as defined in their Windows account. It increases your chances of preventing unwanted access to or behavior of your application
- **Code access security:** It assigns permissions to assemblies based on assembly evidence. Code access security uses the location from which executable code is obtained and other information about the identity of code as a primary factor in determining what resources the code should have access to

10.10 REVIEW QUESTIONS

1. What is the Code Access Security?
2. Explain the .Net security tools and how do we use those tools?
3. Explain the Role Based Security.
4. Define the Type safety.

10.11 FURTHER READINGS

CLR via C#, Second Edition (Paperback) by Jeffrey Richter, Publisher: Microsoft Press; 2nd edition (February 22, 2006)

Essential C# 2.0 (Microsoft .NET Development Series) (Paperback) by Mark Michaelis, Publisher: Addison-Wesley Professional (July 13, 2006)

C# Programming Language, The (2nd Edition) (Microsoft .NET Development Series) (Hardcover) by Anders Hejlsberg, Scott Wiltamuth, Peter Golde, Publisher: Addison-Wesley Professional; 2 edition (June 9, 2006)

Programming in the Key of C# (Paperback) by Charles Petzold, Publisher: Microsoft Press; 1 edition (July 30, 2003)