UNIT I INTRODUCTION TO C#

Introducing C# - Introduction to .Net framework and Architecture - Understanding .NET-Overview of C#- Literals- Variables-Constant Variables – Scope of Variables – Boxing and Unboxing - Data Types- Operators- Expressions- Branching- Looping- Methods- Arrays- Strings- Structures- Enumerations.

.NET Environment Unit 1

What is .NET ?

. Net is a new framework for developing windows-based and web-based applications within the Microsoft environment.

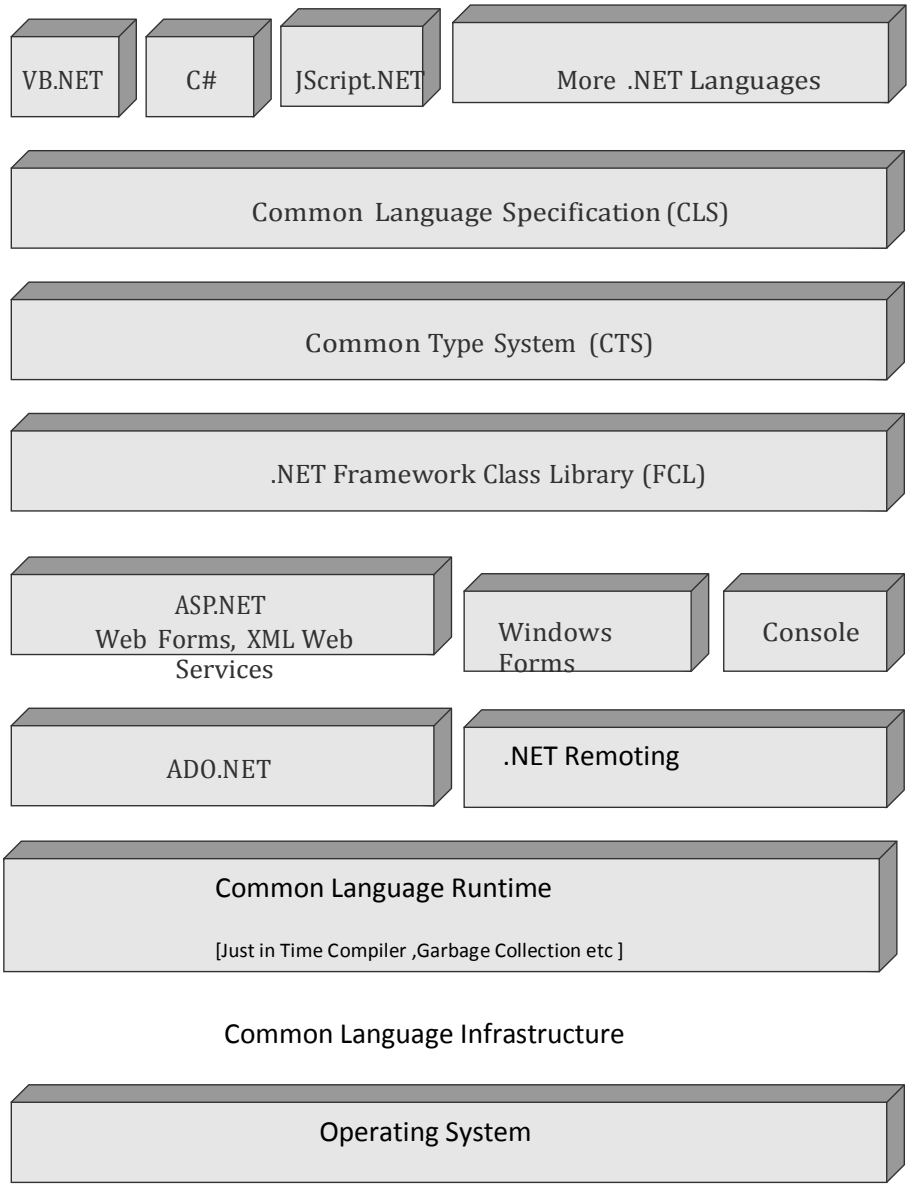
Why .Net?

.NET is a Multilanguage and multiplatform operating environment. Compare this to Java, which is single-language and multiplatform. .NET offers C#, Visual Basic .NET, and many more .NET-compliant languages.

.

.NET Framework Architecture

. .NET is tiered, modular, and hierarchal. Each tier of the .NET Framework is a layer of abstraction. .NET languages are the top tier and the most abstracted level. The common language runtime is the bottom tier, the least abstracted, and closest to the native environment. This is important since the common language runtime works closely with the operating environment to manage .NET Applications. The .NET Framework is partitioned into modules, each with its own distinct responsibility. Finally, since higher tiers request services only from the lower tiers, .NET is hierarchal. The architectural layout of the .NET Framework is illustrated in the following Fig



1) COMMON LANGUAGE INFRASTRUCTURE

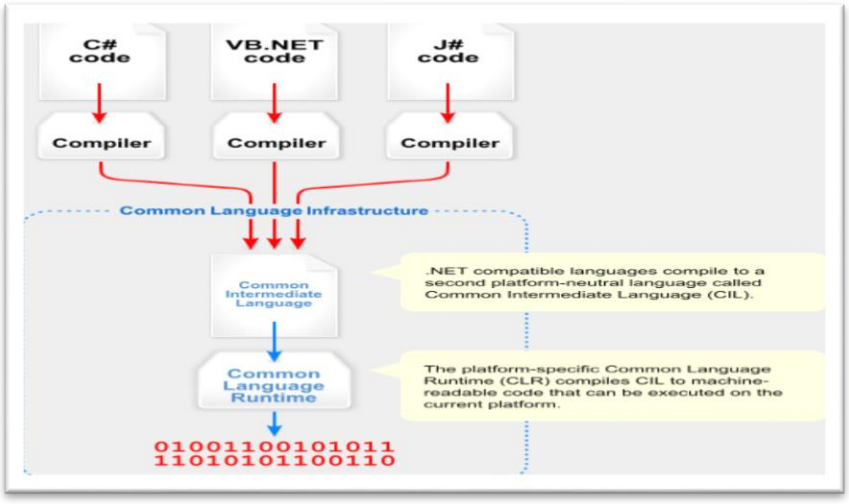
Common Language Infrastructure (CLI) is a Microsoft specification for running high-level language program applications in different computer systems without changing the applicationcode.

CLI is based on the Microsoft .NET concept that some high-level language programs require modifications due to system hardware and processing constraints. CLI compiles applications as Intermediate Language (IL), which is automatically compiled as native system code. This approach allows applications to run without code rewrites in limited systems.

C#, VB.NET, J# etc are some of the .NET languages which has its own compiler and compiles to a Common Intermediate Language. This is also called as Microsoft Intermediate language or Intermediate language code. The CIL is converted as machine readable code by Common language runtime.

Apart from the .net languages it supports third party languages like Fortran,python and perl

Any language which is understood by CLR is called managed code. Language which does not understood by CLR is unmanaged code.In other words MSIL is an managed code



2) COMMON LANGUAGE RUNTIME

Before looking into CLR - let's explain what is meant by runtime. Runtime is an environment in which programs are executed. The **Common Language runtime** is therefore an environment in which we can run our **.NET** applications that have been compiled to IL. This CLR is the core component of the .NET framework which sits on top of the operating system.

The Common Language Runtime is somewhat comparable to the Java Virtual Machine that Sun Microsystems furnishes for running programs compiled from the Javalanguage. Microsoft refers to its Common Language Runtime as a "managed execution environment." A program compiled for the CLR does not need a language-specific execution environment and can easily be moved to and run on any system

Common language runtime manages Jit Compilation, security, code verification, type verification, exception handling, garbage collection, a common runtime, Cross-language integration, Code access security, Object lifetime management, Debugging and profiling support and other important elements of program execution.

Of the many services offered by the common language runtime, we will focus
On the following

- Just in Time Compilation
- Garbage collection
- Code Access security
- Code verification
- Assemblies

Jit compilation

The .NET Framework contains one or more JIT compilers that compile your IL code down to Machine code or code that is CPU-specific.

When Main makes its first call to Writeline the JIT compiler function is called. The JIT compiler function is responsible for compiling a method's IL code into native CPU instructions. Because the IL is being compiled "Just in Time" this component of the CLR is referred as JITter or JIT compiler.

Garbage Collector

The .NET Framework is a garbage-collected environment. Garbage collection is the process of detecting when objects are no longer in use and automatically destroying those objects, thus freeing memory. In .NET, this new garbage collector works so that you as a developer are no longer required to monitor your code for unneeded objects and destroy them. The garbage collector will take care of all this for you. Garbage collection does not happen immediately, but instead the garbage collector will occasionally make a sweep of the heap to determine which objects should be allocated for destruction. This new system completely absolves the developer from hunting down memory usage and deciding when to free memory.

Code Access security

The Common Language Runtime (**CLR**) allows **code** to perform only those operations that the **code** has permission to perform. So CAS is the **CLR**'s **security** system that enforces **security** policies by preventing unauthorized **access** to protected **resources** and **operations**. Using the **Code Access Security**, you can do the following:

- Restrict what your code can do
- Restrict which code can call your code
- Identify code

Code verification

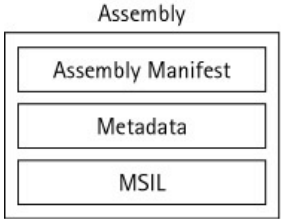
This ensures proper code execution and type safety while the code runs. It prevents source code to perform illegal operations such as accessing invalid memory Locations.

Assemblies

In .NET, a managed application is called an assembly. Without an associated assembly, code will not be able to compile from IL. When you are using the JIT compiler to compile your code from managed code to machine code, the JIT compiler will look for the IL code that is stored in a portable executable (PE) file along with the associated assembly manifest.Every time you build a Web Form or Windows Form application in .NET, you are actually building an assembly. Every one of these applications will contain at least one assembly.As in the Windows DNA world where DLLs and EXEs are the building blocks of applications, in the .NET world, it is the assembly that is the used as the foundation of applications.

The structure of an assembly

Assemblies are made up of the following parts:



Assembly manifest: This is where the details of the assembly are stored. The assembly is stored within the DLL or EXE itself. Assemblies can either be single or multi file assemblies and, therefore, assembly manifests can either be stored in the assembly or as a separate file. The assembly manifest also stores the

version number of the assembly to ensure that the application always uses the correct version. When you are going to have multiple versions of
an assembly on the same machine, it is important to label them carefully so that the CLR
knows which one to use.

Metadata: This is basically "data about data" or a description of the contents of a .NET component. This metadata is stored within the assembly manifest.

MSIL: The compiler translates the source code into Microsoft intermediate language (MSIL), which is a CPU-independent set of instructions that can be efficiently converted to native code.

3)ADO.NET

ADO.NET (ActiveX Data Objects for .NET) is a set of computer software components that programmers can use to access data and data services. It is a part of the base class library that is included with the Microsoft .NET Framework. It is commonly used by programmers to access and modify data stored in relational database systems, though it can also access data in non-relational sources. ADO.NET is an exceptional and a worthy successor to ADO. ADO.NET offers managed providers for Microsoft SQL and OLE DB databases and is a set of managed classes in the System.Data namespace. The System.Data.SqlClient namespace contains classes related to Microsoft SQL, while System.Data.OleDb encompasses classes pertaining to OLE DB providers.

4) .NET Remoting

.NET Remoting is similar to Web services conceptually. However, with .NET Remoting the developer chooses the transmission protocol, data protocol, data port, and other aspects of the remoting architecture necessary to open a channel for client-server communication. In essence, a developer is setting the specifications of the remoting infrastructure. In this way, .NET Remoting offers unlimited possibilities. Like Web services, .NET Remoting can leverage open standards, such as XML, HTTP, and SMTP. .NET Remoting is fully extensible. Custom or and proprietary standards can also be plugged in. The namespace for .NET Remoting is System.Runtime.Remoting.

5) ASP.NET

ASP.NET is used to create dynamic Web applications and is the successor to ASP. While IIS 5 and 6 support side-by-side execution of ASP and ASP.NET,ASP.NET uses ADO.NET, server-side controls, and other techniques to promote a highly distributed and scalable model. Also, ASP.NET hosts Web applications in application domains within the worker process (aspnet_wp.exe) to heighten performance and lower overhead. Finally, ASP.NET uses compiled pages instead of interpreted pages to
improve performance.

6) XML web SERVICES

Web services are the basis of the programmable Web and distributed applications that transcend hardware and operating environments. Web services are not unique to Microsoft. Microsoft, IBM, Sun, and other vendors are promoting Web services as integral components of their recent initiatives. Web services promote remote function calls over the Internet. A Web service exposes functionality to the entire world—any device at any time. Anyone with Web-enabled software, such as a browser, that understands HTML and HTTP can access a Web service. Any device, large or small, that is Web enabled can access a Web service. Visual Studio .NET removes the challenge of creating a Web service. Developers can create Web services with limited or no knowledge of SOAP, XML, or WSDL. The first step is to create an ASP.NET Web service project. The new project is a starter kit for a Web service application. A Web service class, sample Web method, web.config file, global.asx file, and the remaining plumbing of a Web service are provided.

7) WINDOWS FORMS AND CONSOLE APPLICATIONS

Windows Forms is the form generator for client-side applications and is similar to the forms engine of Visual Basic 6. Visual Basic programmers using VB.NET will be familiar with the look and feel of Windows Forms, but this similarity is largely cosmetic and there are substantial differences in the implementation. In .NET, console applications are available to all managed languages. Console applications are useful for logging, instrumentation, and other text-based activities.

8) .NET FRAMEWORK CLASS LIBRARY

The .NET Framework Class Library (FCL) is a set of managed classes that provide access to system services. Once you have learned any .NET language, you have learned 40 percent of every other managed language. The same classes, methods, parameters, and types are used for system services regardless of the language. This is one of the most important contributions of FCL.
Here is the C# version of the program

```
//C#Program
using System;
  class Program
  {
     static void Main()
     {
        int i,j,k;
     i = 10;
     j = 5;
     k = i + j;
     Console.WriteLine("sum of {0} and {1} is {2}", i, j, k);
    }
}
```

Next is the VB.NET version of the program.

```
    Module Module1
      Sub Main()
        Dim i, j, k As Integer
        i = 10
        j = 5
        k = i + j
        Console.WriteLine("sum of {0} and {1} is {2}", i, j, k)
      End Sub
    End Module
```

Both versions of the program are nearly identical. The primary difference is that C# uses semicolons at the end of statements, while VB.NET does not. The syntax and use of the Console class are identical for both the programs.

FCL includes some 600 managed classes. A flat hierarchy consisting of hundreds of classes would be difficult to navigate. Microsoft partitioned the managed classes of FCL into separate namespaces based on functionality. For example, classes pertaining to local input/output can be found in the namespace System. IO. To further refine the hierarchy, FCL namespaces are often nested; the tiers of namespaces are delimited with dots. System.Runtime.InteropServices, System.Security.Permissions, and System.Windows.Forms are examples of nested namespaces. The root namespace is System, which provides classes for console input/output, management of application domains, delegates, garbage collection, and more.

NAMESPACE : NameSpace is the Logical group of types or we can say namespace is a container (e.g Class, Structures, Interfaces, Enumerations, Delegates etc.), example System.IO logically groups input output related features , System.Data.SqlClient is the logical group of ado.net Connectivity with Sql server related features.

Namespace	Purpose of Types
System	All the basic types used by every application.
System.Collections	Managing collections of objects. Includes the popular collection types such as Stacks, Queues, Hashtables, and so on.
System.Drawing	Manipulating 2D graphics. Typically used for Windows Forms applications and for creating images that are to appear in a web form.
System.IO	Doing stream I/O, walking directories and files.

You should be aware, however, that in addition to supplying the more general namespaces, the FCL offers namespaces whose types are used for building specific application types. The table below lists some of the application-specific namespaces:

Namespace	Purpose of Types
System.Web.Services	Building web services.
System.Web.UI	Building web forms.
System.Windows.Forms	Building Windows GUI applications.
System.ServiceProcess	Building a Windows service controllable by the Service Control Manager.

9)COMMON TYPE SYSTEM

The Common Type System defines how data types are declared, used, and managed in the runtime, and is also an important part of the runtime's support for the Cross-Language Integration. The common type system performs the following functions:

- Establishes a framework that enables cross-language integration, type safety, and high performance code execution.
- Provides an object-oriented model that supports the complete implementation of many programming languages.
- Defines rules that languages must follow, which helps ensure that objects written in different languages can interact with each other.

One important reason that strong data typing is important is that, if a class is to derive from or contains instances of other classes, it needs to know about all the data types used by the other classes. Indeed, it is the absence of any agreed system for specifying this information in the past that has always been the real barrier to inheritance and interoperability across languages. This kind of information is simply not present in a standard executable file or DLL.

Suppose that one of the methods of a VB.NET class is defined to return an Integer – one of the standard data types available in VB.NET. C# simply does not have any data type of that name. Clearly, we will only be able to derive from the class, use this method, and use the return type from C# code if the compiler knows how to map VB.NET's Integer type to some known type that is defined in C#. So how is this problem circumvented in .NET?

This data type problem is solved in .NET through the use of the **Common Type System** (**CTS**). The CTS defines the predefined data types that are available in IL, so that all languages that target the .NET framework will produce compiled code that is ultimately based on these types. For the example that we were considering before, VB.NET's Integer is actually a 32-bit signed integer, which maps exactly to the IL type known as Int32. This will therefore be the data type specified in the IL code. Because the C# compiler is aware of this type, there is no problem. At source code level, C# refers to Int32 with the keyword int, so the compiler will simply treat the VB.NET method as if it returned an int.The CTS doesn't merely specify primitive data types, but a rich hierarchy of types, which includes well defined points in the hierarchy at which code is permitted to define its own types.

10)COMMON LANGUAGE SPECIFICATION

Common Language specification is a set of base rules to which any language targeting the CLI should conform in order to interoperate with other CLS-compliant languages. The CLS rules define a subset of the Common Type System

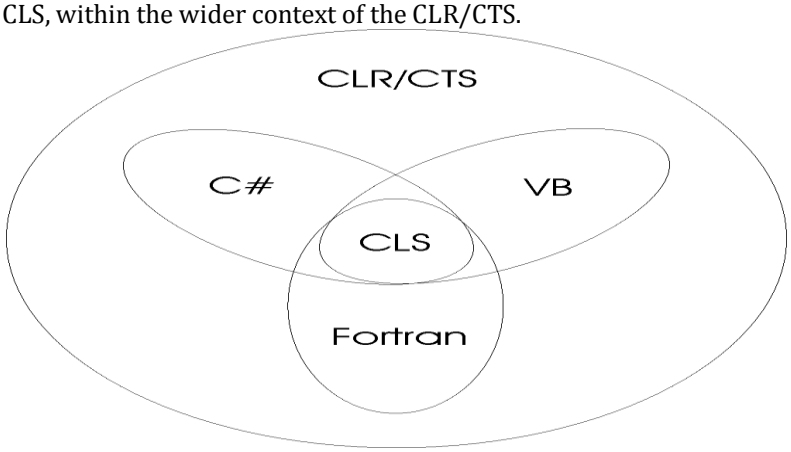
To fully interact with other objects regardless of the language they were implemented in, objects must expose to callers only those features that are common to all the languages they must interoperate with. For this reason, the Common Language Specification (CLS), which is a set of basic language features needed by many applications, has been defined.

COM allows objects created in different languages to communicate with one another. The CLR goes further. It integrates all languages to let objects created in one language be treated as equal citizens by code written in a completely different language. To make this possible, the CLR defines a standard behavior for types, embeds self-describing type information (metadata), and provides a common

execution environment. Language integration is a fantastic goal, of course, but the truth of the matter is that programming languages are very different from one another. For example, some languages lack features commonly used in other languages:

- case-sensitivity
- unsigned integers
- operator overloading
- methods that support a variable number of parameters

If you intend to create types that are easily accessible from other programming languages, then it is important that you use only features of your programming language that are uaranteed to be available in all other languages. To help you with this, Microsoft has defined a *common language specification* (CLS) that details for compiler vendors the minimum set of features that their compilers must support if they are to target the runtime. Note that the CLR/CTS supports a lot more features than the subset defined by the common language specification, so if you don't care about language interoperability, you can develop very rich types limited only by the capabilities of the language. Specifically, the CTS defines rules to which externally visible types and methods must adhere if they are to be accessible from any CLR-compliant programming language. Note that the CLS rules do not apply to code that is only accessible within the defining assembly. The figure below summarizes the way in which language features overlap with the CLS, within the wider context of the CLR/CTS.



As this figure shows, the CLR/CTS offers a broadly inclusive set of features. Some languages expose a large subset of the CLR/CTS; in fact, A programmer willing to write in IL assembly language is able to use all the features offered by the CLR/CTS. Most other languages—such as C#, VB, and Fortran—expose a subset of the CLR/CTS features to the programmer. The CLS defines a minimum set of features that all languages must support.If you are designing a type in one language and you expect that type to be used by another language, then you should not take advantage of any features that are outside of the CLS. Doing so means that your type members might not be accessible by programmers writing code in other programming languages.

OVERVIEW OF C#

A sample program in c#

```
using System;
class Program
{
   static void Main()
   {
      Console.WriteLine("Welcome to c#");
   }
}
```

The above program prints Welcome to c# in the console

- Every C# program will have a Main method which is the entry point of a program.
- Main method will always be static and returns void.It will be defined within a class.
- C# is case sensitive.
- If multiple mains are vailable only one main method will be invoked first.
- Console.writeline is used to print the values. The above program prints Welcome to c# in the console
- System is the namespace. If there are any calls to Console.WriteLine, the System namespace is required. Many other common functions require System as well.The syntax is

  using System;

LITERALS

Literals are value constants assigned to variables in a program.

```
bool result = true;
char capitalC = 'C';
byte b = 100;
short s = 20000;
int i = 300000;
```

In the above example, literals are **true**, **'C'**, **100**, **20000** and **300000**.

Types of Literals

In C# language, there are several types of literals:

- Boolean
- Integer
- Real
- Character
- String

Boolean Literals

Boolean literals are:

- **true**
- **false**

When we assign a value to a variable of type **bool** we can use only one of these two values or a Boolean expression (which is calculated to **true** or **false).**

Boolean Literals – Example

Here is an example of a declaration of a variable of type **bool** and assigning a value, which represents

the Boolean literal **true**:

```
bool result = true;
```

Integer Literals

Integer literals are **sequences of digits**, a sign (**+**, **-**), suffixes and prefixes. Using prefixes we can present integers in the

Integer Literals – Examples

```
// The following variables are initialized with the same value
int numberInDec = 16;
int numberInHex = 0x10;
```

Real Literals

Real literals are a **sequence of digits**, a sign (**+**, **-**), suffixes and the decimal point character. We use them for values of type **float**, **double** and **decimal**. Real literals can be represented in exponential format

Real Literals – Examples

Here are some examples of real literals' usage:

```
// The following is the correct way of assigning a value:
float realNumber = 12.5f;
// This is the same value in exponential format:
realNumber = 1.25e+1f;
// The following causes an error, because 12.5 is double
float realNumber = 12.5;
```

Character Literals

Character literals are **single characters enclosed in apostrophes** (single quotes). We use them to set the values of type **char**. The value of a character literal can be:
a character, for example **'A'**;

String Literals

String literals are used for data of type **string**. They are a sequence of characters enclosed in double quotation marks.

String Literals – Examples

```
string quotation = "Hello";
```

VARIABLES

A variable is an identifier that denotes a storage location used to store a data value.
Variable name may consists of albhabets,digits,uderscore under following conditions

- They must not begin with a digit
- Uppercase and lowercase are distinct .The variable total is
- not same as TOTAL
- It should not be keyword
- variable name can be of any length

Declaration of variables

Declaration does three things

- It tells the compiler what the variable name is
- It specifies what type of data variable hold
- The place of declaration decides the scope of variable

Constant variables

- The variables whose value do not change during the execution of program is constant.
- After declaration of constant it should not be assigned any value
- Constants cannot be declared inside a method.It should be class level

Constant variables examples

```
Const int rows =10
Const int cols=20
```

BOXING AND UNBOXING

Boxing means conversion of value type on stack to a object type on heap

```
int m=100;
object om=m;  //creates a box to hold m
```

- Boxing operation creates a copy of the m integer to object om.
- Both variables m and om exist but the value of om resides on the heap
- It also means tha the values are independent of each other

Unboxing is the process of converting object type to value type

```
Int m = 10
Object om = m; //box m
Int n =(int)om   //unbox
```

We can only unbox which has previously boxed

DATA TYPES

Data types are sets (ranges) of values that have similar characteristics. Every variable in c# is associated with a data type. Data type specify the size and type of values that stored

short	16 bit
int	32 bit
long	64 bit
float	32 bit
double	64 bit
char	Single character
String	Set of character
Boolean	True or false

OPERATORS

Operators in C# can be separated in several different categories:

- **Arithmetic** operators – they are used to perform simple mathematical operations.
- **Assignment** operators – allow assigning values to variables.
- **Comparison** operators – allow comparison of two literals and/or variables.
- **Logical** operators – operators that work with Boolean data types and Boolean expressions.
- **Binary** operators – used to perform operations on the binary representation of numerical data.
- **Type conversion** operators – allow conversion of data from one type to another.

Operator Categories

Below is a list of the operators, separated into categories:

Category	**Operators**
arithmetic	**-, +, *, /, %, ++, --**
logical	**&&, ||, !, ^**
binary	**&, |, ^, ~, <<, >>**
comparison	**==,!=, >, <, >=, <=**
assignment	**=, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=**
string	**+**

concatenation	
type conversion	**(type), as, is, typeof, sizeof**
other	**., new, (), [], ?:, ??**

EXPRESSIONS

Expressions are sequences of operators, literals and variables that are calculated to a value of some type (number, string, object or other type). Here are some examples of expressions:

```
int r = (150-20) / 2 + 5;

// Expression for calculating the surface of the circle
double surface = Math.PI * r * r;

// Expression for calculating the perimeter of the circle
double perimeter = 2 * Math.PI * r;
Console.WriteLine(r);
Console.WriteLine(surface);
Console.WriteLine(perimeter);
```

BRANCHING AND LOOPING

In normal circumstances whenever a computer runs a program, execution starts from the first line of code to the last. With the help of control statements you can change the computer's control of executing the program automatically reading the next line of code to reading a different line of code.

Following are the control statements available in C#

- if else
- Nested if else
- For Loop
- Wile Loop
- Switch case

If Else Statement

The if statement implements conditional execution. Following is the syntax of control statement.

SYNTAX:

```
If (Expression)
  {
      statement1
  }
else
  {
      statement2
  }
```

In the above structure if the expression returns true, this will execute statment1, if the expression evaluates to false it executes the statment2.

If else example

```
class program
{
  static void Main()
```

```
        {
        int length =5;
        int breadth = 10;
        if (length == breadth)
        {
                Console.WriteLine("It is a Square");
           }
        else
           {
          Console.WriteLine("It is not a  Square");
           }
  }
}
```

Nested if else

Following is the general structure of nested if else.

SYNTAX:

```
if  ( condition1)
  {
        if (condition2)
           {
           Statement1
           }
       else
           {
           Statement2
           }
  }
 else
   {
     Statement3
   }
```

In the above structure, if the condition1 fails then statment3 gets executed. If condition1 returns true then condition2 is evaluated. If condition2 returns true then statment2 is executed else statement3 is executed.

For loop statement

The for loop is executes the body of the loop as long as the condition is true.First the variables is initialised,then the condition is checked and finally the value of initialisation expression is incremented.The for loop is good for situatuin where we know exactly how many times the statements within the body of the loop to be executed

SYNTAX:

```
for ( initialisation ; condition ; increment expression)

        {
                Statements
        }
```

In the above structure, the initialisation expression is a list of expression separated by commas. There can be one or multiple expression depending upon the requirement. It is mandatory to initialise the variable in a for loop. Then if the condition in the for loop returns true then the statements in the for loop is executed. If it returns false then these statements are skipped.

Once the statements in the for loop is executed, the control first transfers to the for loop's incremental expression for incrementing the value of the control variable. Again in the for loop condition is evaluated against the new value of control variables and the process continues until the conditional expression becomes false

For loop example

```
class program
{
        static void Main()
        {
    int len;
          for ( len = 0; len< 4; len++)
           {
                Console.WriteLine (" All the best");
     }
  }
}
```

The above for loop executes four times.

```
class program
{
        static void Main()
        {
    int len;
          for ( len = 1; len< 4; len++)
           {
                Console.WriteLine (" All the best");
     }
  }
}
```

The above for loop executes three times as i value starts from 1.

While Loop

The while loop is the simplest of all the loops.It is very much similar to the for loop.The general structure of whle loop is as follows

SYNTAX:

```
while ( condition)
        {
                Statements
        }
```

In while loop,first the expression is evaluated and if it returns true,then the statements are executed.Again the control transfers to the expression and evaluates it.If it reutns true once again the same statements are executed.This whole process is repeated until the expression returns false.

While Loop example

```
class program
{
        static void Main()
        {
          int x = 0;
          while (x <  3 )
```

```
            {
                    Console.WriteLine (" All the best");
                    x++;
      }
   }
}
```

The above while loop executes 3 times.

Switch case statement

When we use if statemnt to control the selection among a number of choices in program,but it sometimes increases the complexity of a program.Switch statement is another alternative for making selections in easier way.the general structure of switch statmetn is as follows.

SYNAX

```
switch( expression)
    {
       case value1 :
                statements
                break;
         case value2:
                statements
                break;
          case value2:
                statements
                break;
          default;
                statements
                break;
      }
```

The switch statement checks the value of the expression against the list of case values and when a match is found, a block of statements associated with that case is executed. Following are the important points to remember in switch case

- An expression may contain integer, character or string type values
- Values can be constant or constant expressions
- Statements can contain zero or multiple statements.
- The keyword break tells that this this is the end of the corresponding case and causes an exit from the switch statement, transferring the control to the statement that is followed by switch statement.
- When the value of the expression does not meet any case then default statements are executed

Switch case example

```
        class Program
  {
    static void Main()
     {
       Console.WriteLine("Enter 1 for Red, 2 For Green 3 For Blue");
        int col = Console.Read() ;
                switch (col)
                      {
                        case 1 :
               Console.WriteLine("The color is Red");
```

```
                    break;
                      case 2:
                                Console.WriteLine("The color is Green");
                    break;
                      case 3 :
                      Console.WriteLine("The color is Blue");
                    break;
            }
      }
  }
```

In the above example Console.Read() is used to get the input from the user.When input given as 1 then case 1 is executed.

METHODS

A **method** is a basic part of a program. It can **solve a certain problem, eventually take parameters and return a result**.

In the C# language, a method can be declared only between the opening "{" and the closing "}" brackets of a class. A typical example for a method is the already known method **Main(…)** – that is always declared between the opening and the closing curly brackets of our class. An example for this is shown below:

```
public class HelloCSharp
{ // Opening brace of the class

// Declaring our method between the class' body braces
static void Main(string[] args)
{
Console.WriteLine("Hello C#!");
}

} // Closing brace of the class
```

Method Declaration

To **declare a method** means to **register** the method in our program. This is shown with the following declaration:

[static] <return_type> <method_name>([<param_list>])

There are some mandatory elements to declare method:

- Type of the result, returned by the method – **<return_type>**.
- Method's name – **<method_name>**.
- List of parameters to the method – **<param_list>** – it can be empty list or it can consist of a sequence of parameters declarations.

To clarify the **elements of method's declaration**, we can use the **Main(…)** method from the example **HelloCSharp** show in the previous block:

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

As can be seen the **type of returned value** is **void** (i.e. that method does not return a result), the method's name is **Main**, followed by round brackets, between which is a list with the method's **parameters**. In the particular example it is actually only one **parameter** – the array **string[] args**

The sequence, in which the elements of a method are written, is strictly defined. Always, at the very first place, is the type of the value that method returns **<return_type>**, followed by the method's name

<method_name> and list of parameters at the end **<param_list>** placed between in round brackets – "**(**" and "**)**". Optionally the declarations can have **access modifiers** (as **public** and **static**).

When a method is declared keep the sequence of its elements description: first is the type of the value that the method returns, then is the method's name, and at the end is a list of parameters placed in round brackets.

The list with parameters is allowed to be **void** (empty). In that case the only thing we have to do is to type "**()**" after the method's name. Although the method has not parameters the round brackets must follow its name in the declaration. For now we will not focus at what **<return_type>** is. For now we will use **void**, which means the method will not return anything. Later, we will see how that can be changed

Implementation (Creation) of Own Method

After a method had been declared, we must write its implementation. As we already explained above, **implementation (body)** of the method consists of the code, which will be executed by calling the method. That code must be placed in the method's body and it represents the method's logic.

The Body of a Method

Method body we call the piece of code, that is placed in between the curly brackets "**{**" and "**}**", that directly follow the method declaration.

```
static <return_type> <method_name>(<parameters_list>)
{
// … code goes here – in the method's body …
}
```

The real job, done by the method, is placed exactly in the method body. So, the algorithm used in the method to solve the particular task is placed in the method body.

```
static void PrintLogo()

{ // Method's body starts here

Console.WriteLine("Microsoft");
Console.WriteLine("www.microsoft.com");

} // … And finishes here
```

Invoking a Method

Invoking or **calling a method** is actually the process of **execution** of the method's code, placed into its body.

It is very easy to invoke a method. The only thing that has to be done is to write the method's name **<method_name>**, followed by the round brackets and semicolon "**;**" at the end:

```
<method_name>();
```

Method Declaration and Method Invocation

In C# the order of the methods in the class is not important. We are allowed to invoke (call) a method before it is declared in code:

```
static void Main()
{
// …
PrintLogo();
// …
```

```
}
static void PrintLogo()
{
Console.WriteLine("Microsoft");
Console.WriteLine("www.microsoft.com");
}
```

Declaring Methods with Parameters

To pass information necessary for our method we use the **parameters list**. As was already mentioned, we must place it between the brackets following the method name, in method the declaration:

```
static <return_type> <method_name>(<parameters_list>)
{
// Method's body
}
```

The parameters list **<parameters_list>** is a list with zero or more **declarations of variables**, separated by a comma, so that they will be used for the implementation of the method's logic:

ARRAYS

Arrays are vital for most programming languages. They are collections of variables, which we call **elements**

An array's elements in C# are numbered with 0, 1, 2, … N-1. Those numbers are called **indices**. The total number of elements in a given array we call **length of an array**. All elements of a given array are of the same type, no matter whether they are **primitive** or **reference** types. This allows us to represent a group of similar elements as an ordered sequence and work on them as a whole. Arrays can be in different dimensions, but the most used are the **one-dimensional** and the **two-dimensional** arrays. One-dimensional arrays are also called **vectors** and two-dimensional are also known as **matrices**

Declaring an Array

We declare an array in C# in the following way:

```
int[] myArray;
```

In this example the variable **myArray** is the name of the array, which is of integer type (**int[]**). This means that we declared an array of integer numbers. With **[]** we indicate, that the variable, which we are declaring, is an array of elements, not a single element.

Creation of an Array – the Operator "new"

In C# we create an array with the help of the keyword **new**, which is used to allocate memory:

```
int[] myArray = new int[6];
```

Array Initialization and Default Values

In C# all variables, including the elements of arrays have a **default initial value**. This value is either **0** for the numeral types or its equivalent for the non-primitive types (for example **null** for a reference type and **false** for the **bool** type).

Of course we can set initial values explicitly. We can do this in different ways. Here is one of them:

```
int[] myArray = { 1, 2, 3, 4, 5, 6 };
```

Declaration and Initialization of an Array – Example

```
string[] daysOfWeek =
{ "Monday", "Tuesday", "Wednesday","Thursday", "Friday", "Saturday", "Sunday" };
```

Access to the Elements of an Array

We access the array elements directly using their **indices**. Each element can be accessed through the name of the array and the element's **index** (consecutive number) placed in the brackets. We can access given elements of the array both for reading and for writing, which means we can treat elements as variables.

Here is an example for accessing an element of an array:

```
myArray[index] = 100;
```

In the above example above we set a value of 100 to the element, which is at position **index**.

```
int[] myArray = new int[6];
myArray[1] = 1;
myArray[5] = 5;
```

Reading an Array from the Console

```
for (int i = 0; i < n; i++)
{
array[i] = int.Parse(Console.ReadLine());
}
```

Array example : finding largest array element & average of Array elements via methods.

```
using System;
class ArrayFunction
{
public static void Main()
{
long Largest;
double Average;
int c;
int num;
int[] array1;
Console.Write("Enter the number of Elements in an Array : ");
c=int.Parse(Console.ReadLine());
array1=new int[c];
for (int i=0 ; i>c ;i++)
{
Console.WriteLine("Enter the element " + i);
num=int.Parse(Console.ReadLine());
array1[i]=num;
}
foreach (int i in array1)
{
Console.Write(" " + i);
}
Console.WriteLine ();
Largest = Large(array1);
Average = Avg(array1);
Console.WriteLine ("\n The largest element in the array is " +
Largest);
Console.WriteLine ("The Average of elements in the array is " +
Average);
Console.ReadLine();
}
// Determining the largest array element
static int Large (params int [] arr)
{
int temp=0;
for ( int i = 0; i < arr.Length; i++)
{
```

```
if (temp <= arr[i])
{
temp = arr[i];
}
}
return(temp);
}

// Determining the average of array elements
static double Avg (params int [] arr)
{
double sum=0;
for ( int i = 0; i < arr.Length; i++)
{
sum = sum + arr[i];
}
sum = sum/arr.Length;
return(sum);
}
}
```

Output:

```
Enter the number of Elements in an Array : 5
Enter the element 1 : 5
Enter the element 2 : 7
Enter the element 3 : 3
Enter the element 4 : 1
Enter the element 5 : 8
largest element in the array is 8
The Average of elements in the array is 4.8
```

ARRAYS OF ARRAYS [JAGGED ARRAY]

In C# we can have arrays of arrays, which we call **jagged** arrays. Jagged arrays are **arrays of arrays**, or arrays in which each row contains an array of its own, and that array can have length different than those in the other rows.

Declaration and Allocation an Array of Arrays

The only difference in the declaration of the jagged arrays compared to the regular multidimensional array is that we do not have just one pair of brackets. With the jagged arrays we have a pair brackets per dimension. We **allocate** them this way:

```
int[][] jaggedArray;
jaggedArray = new int[2][];
jaggedArray[0] = new int[5];
jaggedArray[1] = new int[3];
```

Here is how we declare, allocate and initialize an array of arrays (a jagged array whose elements are arrays of integer values):

```
int[][] myJaggedArray = {
new int[] {5, 7, 2},
new int[] {10, 20, 40},
new int[] {3, 25}
};
```

Jagged Array Example :

```
using System;
class Program
{
static void Main()
  {
        int[][] jag = new int[3][];
        jag[0] = new int[2];
        jag[0][0] = 11;
        jag[0][1] = 12;
        jag[1] = new int[1] {11};
        jag[2] = new int[3] { 14,15, 16 };
        for (int i = 0; i < jag.Length; i++)
        {
                int[] innerArray = jag[i];
                for (int a = 0; a < innerArray.Length; a++)
                        {
                        Console.WriteLine(innerArray[a] + " ");
                        }
        }
  }
```

STRINGS

A string is a **sequence of characters** stored in a certain address in memory. In the variable of type **char** we can record only one character. Where it is necessary to process more than one character then we use strings .

The System.String Class

The class **System.String** enables us to handle **strings in C#**. For declaring the strings we will continue using the **keyword string**, which is an alias in C# of the **System.String** class from .NET Framework. The work with string facilitates us in manipulating the text content: construction of texts, text search and many other operations. Indexing, as it is used in arrays, takes indices from **0** to **Length-1**.

Example of **declaring a string**:

```
string greeting = "Hello, C#";

string str = "abcde";
char ch = str[1]; // ch == 'b'
str[1] = 'a'; // Compilation error!
ch = str[50]; // IndexOutOfRangeException
```

Strings – Simple Example

```
string message = "This is a sample string message.";
Console.WriteLine("message = {0}", message);
Console.WriteLine("message.Length = {0}", message.Length);
for (int i = 0; i < message.Length; i++)
{
Console.WriteLine("message[{0}] = {1}", i, message[i]);
}


// Console output:
// message = This is a sample string message.
// message.Length = 31
```

```
// message[0] = T
// message[1] = h
// message[2] = i
// message[3] = s
        // …
```

Strings Operations

Comparison for Equality

If the requirements are to **compare the two strings** in order to determine whether their values are **equal or not**, the most convenient method is the **Equals(…)**, which works equivalently to the **operator ==**.

```
string word1 = "C#";
string word2 = "c#";
Console.WriteLine(word1.Equals("C#"));
Console.WriteLine(word1.Equals(word2));
Console.WriteLine(word1 == "C#");
Console.WriteLine(word1 == word2);
// Console output:
// True
// False
// True
// False
```

Strings Concatenation

Gluing two strings and obtaining a new one as a result is called **concatenation**. It could be done in several ways: through the method **Concat(…)** or with the operators **+** and **+=**.

Example

```
string greet = "Hello, ";
string name = "reader!";
string result = string.Concat(greet, name);
```

```
string greet = "Hello, ";
string name = "reader!";
string result = greet + name;
```

Switching to Uppercase and Lowercase Letters

```
string text = "All Kind OF LeTTeRs";
Console.WriteLine(text.ToLower());

// all kind of letters
```

Finding All Occurrences of a Substring – Example

Sometimes we want to **find all occurrences of a particular substring within another string**

```
string quote = "The main intent of the \"Intro C#\"" +
" book is to introduce the C# programming to newbies.";
string keyword = "C#";
int index = quote.IndexOf(keyword);
while (index != -1)
```

```
{
Console.WriteLine("{0} found at index: {1}", keyword, index);
index = quote.IndexOf(keyword, index + 1);
}


string path = "C:\\Pics\\CoolPic.jpg";
string fileName = path.Substring(8, 7);
```

STRUCTURES- ENUMERATIONS

Predefined types – int,double etc
User defined types – Structure and enumeration

Structures

- Structures are similar to classes in c#
- Structures are used when composite data types are required

Defining a structure

```
struct structname
{
        datamember1
        datamember2
}
```

Example

```
struct student
{
  public string name
  public int rollnumber
}
```

Example program

```
Using system;
Struct item;
{
        Public string name;
        Public int code;
        Public double price;
}
Class structtest
{
        Public static void main()
        {
                Item fan;
                fan.name=”stevejobs”
                fan.code=123;
                fan.price=1576.50;
                console.writeline(“Fan Name:” + fan.name);
                console.writeline(“Fan code:” + fan.code)
                console.writeline(“Fan cost:” + fan.price);
        }
}
```

Enumeration

- Enumeration is a user defined data type
- It provides a way for attaching names to numbers

Enumeration

```
enum shape
{
        circle,
        square,
        triangle
}
```

enum shape {circle,square,triangle}

Circle has the value 0 ,square has the value 1 and triangle has the value 2

An enumeration is a user defined integer type which provides a way for attaching names to numbers thereby increasing the comprehensibility of the code.The enum keyword automotically enumerates a list of words by assigning them 0,1,2, etc

```
Enum shape
{
Circle,
Square,
Triangle
}
```

Enum color[red,blue,Green,Yellow]
Enum position[on,off]
enum day[Sun,Mon,Tue,Wed,Thu,Fri,Sat

2 MARKS

1. Write any four namespaces provided by .Net

   - System
   - System.Web.UI
   - System.Data.SqlClient;
   - System.Text
   - System.Collections

2. What is the purpose of .net framework class library?
   The .NET Framework Class Library (FCL) is a set of managed classes that provide access to system services

3. Name any two features of .Net.
   - Interoperability.
   - Common Language Runtime **engine** (CLR)
   - Language independence.
   - Base **Class** Library.
   - Simplified deployment.
   - **Security**.
   - **Portability**.

4. Expand and define MSIL.
   MSIL: The compiler translates the source code into Microsoft intermediate language (MSIL), which is a CPU-independent set of instructions that can be efficiently converted to native code

5. "Name any two .Net Languages. Write any two third party languages supported by .Net."

   C# , asp.net are the two languages supported by .Net Framework.

   The third party languages supported by .Net are
   Python, Perl

6. What is CLR?List out the services offered by CLR?

7. What is the purpose of Console.ReadLine () ?

   Consol.readline is to get the input from the user

8. How many times the following loop will execute
   ```
   for ( i=1;i<5;i++)
   {
   Console.WriteLine ("Welcome to C#");
   }
   ```

9. What is the output of the following code?
   ```
   String a = "Welcome "
   String b = "Vels University "
   Console.WriteLine ("The value is {1}, {0} ", a, b);"
   ```

10. Write a program to find out whether it is square or not, given its length and breadth.

```
class program
{
static void Main()
        {
        int length =5;
        int breadth = 10;
        if (length == breadth)
{
                Console.WriteLine("It is a Square");
        }
        else
        {
        Console.WriteLine("It is not a  Square");
        }
}
}
```

11. Write the syntax of 'foreach ' statement.

```
foreach(data_type var_name in collection_variable)
{
   // statements to be executed
}
```

12. Differentiate value type from reference type

- **Call by value -> In this case when we call the method of any class (which takes some parameter) from main method using object.Then value of parameter in main method will directly copy to the class method to parameter values respectively. In this case if some changes occurs in values within the method that change not occurs in actual variable .I have full describe this concept through programming which is given below.**
- **Call by reference -> In this case when we call the method,the reference address of variable is passed to the method.If some changes occurs in values within the method that changes occurs in actual variable.To specify this parameter we use 'ref' Keyword at the time of parameter declaration as well as the calling method.**