UNIT –I

❖ What does .NET Represents?

- **NET** stands for **Network Enabled Technology** (Internet).
- In .NET, dot (.) refers to **Object-Oriented,** and NET refers to the internet.
- So, the complete .NET means through Object-Oriented we can implement internet-based applications.

❖ What is a Framework?

- A framework is a software. Or you can say a framework is a collection of many small technologies integrated together to develop applications that can be executed anywhere.

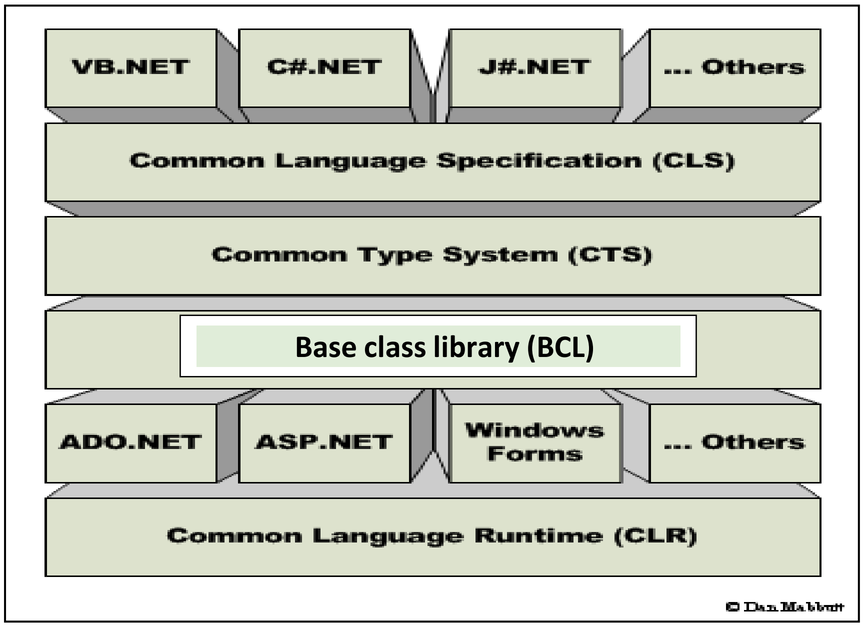
❖ .NET Framework:

- .NET is a framework to develop software applications.
- It is designed and developed by Microsoft and the first beta version released in 2000.
- It is used to develop applications for web, Windows, phone. Moreover, it provides a broad range of functionalities and support.
- The .Net Framework supports more than 60 programming languages such as C#, F#, VB.NET, J#, VC++, JScript.NET, APL, COBOL, Perl, Oberon, ML, Pascal, Eiffel, Smalltalk, Python, Cobra, ADA, etc.

❖ .NET Framework Architecture and Components

The DOT NET Framework provides two things as follows

1. **BCL** (Base Class Libraries)
2. **CLR** (Common Language Runtime)



❖ What is BCL?

- Base Class Libraries (BCL) are designed by Microsoft. Without BCL we can't write any code in .NET.
- So, BCL is also known as the basic building block of .NET Programs. These are installed into the machine when we installed the .NET framework.
- BCL contains pre-defined classes and these classes are used for the purpose of application development.

❖ What is CLR?

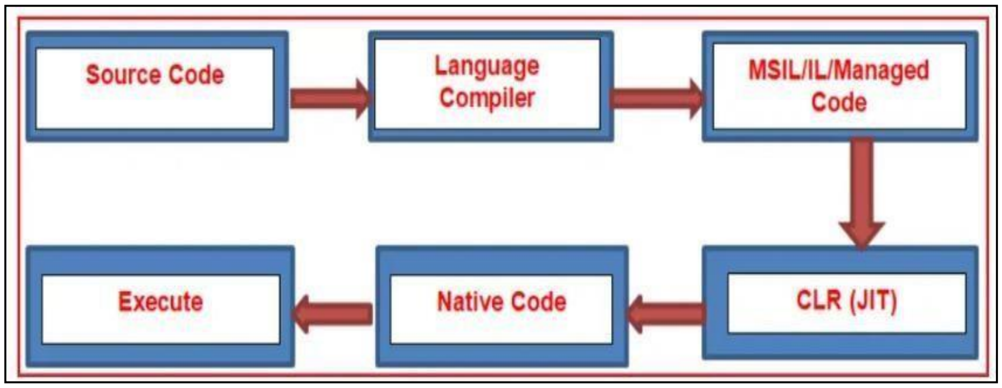
- CLR stands for Common Language Runtime and it is the core component under the .NET framework which is responsible for converting the MSIL (Microsoft Intermediate Language) code into native code.
- **CLR is the heart of the .NET Framework and it contains the following components.**

1. JIT Compiler
2. Memory Manager
3. Garbage Collector
4. Exception Manager
5. Common Language Specification (CLS)
6. Common Type System (CTS)

❖ Working of CLR

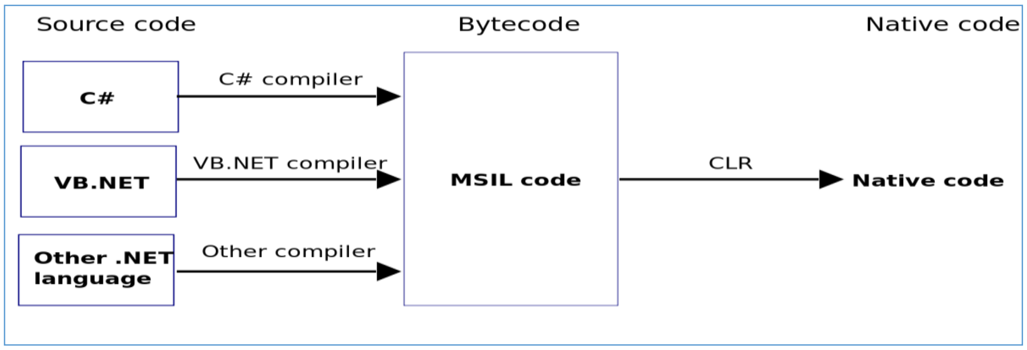
In the .NET framework, the code is compiled twice.

1. In the 1st compilation, the source code is compiled by the respective language compiler and generates the intermediate code which is known as MSIL (Microsoft Intermediate Language) or IL (Intermediate language code), or Managed Code.
2. In the 2nd compilation, MSIL is converted into Native code (native code means code specific to the Operating system so that the code is executed by the Operating System) and this is done by CLR.
3. Always 1st compilation is slow and 2nd compilation is fast.



1. What is JIT?

- JIT stands for the Just-in-Time compiler. It is the component of CLR that is responsible for converting MSIL code into Native Code.
- Native code is code that is directly understandable by the operating system.



What is Intermediate Language (IL) Code in .NET Framework?

❖ The Intermediate Language or **IL code** in .NET Framework is a half-compiled or partially-compiled or CPU-independent partially compiled code and
❖ This code cannot be executed by Operating System.

2. Memory Manager:

- The Memory Manager component of CLR in the .NET Framework allocates the necessary memory for the variables and objects that are to be used by the application.

3. Garbage Collector:

- When a dot net application runs, lots of objects are created. At a given point in time, it is possible that some of those objects are not used by the application.
- So, Garbage Collector in .NET Framework is nothing but a Small Routine or you can say it's a Background Process Thread that runs periodically and try to identify what objects are not being used currently by the application and de-allocates the memory of those objects.

4. Exception Manager:

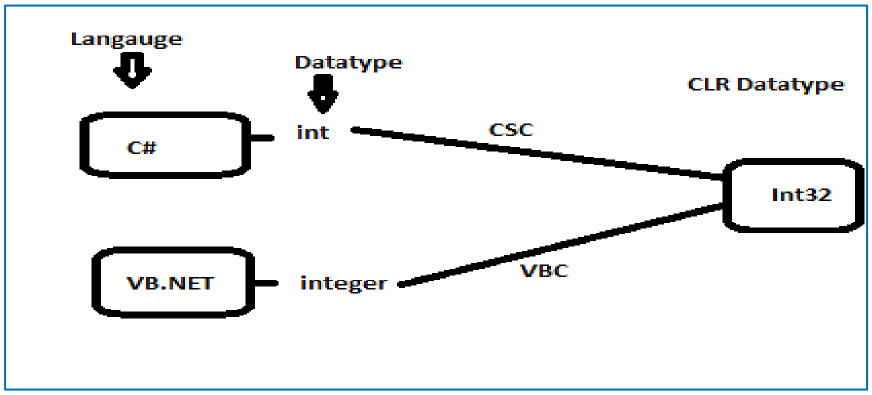
- The Exception Manager component of CLR in the .NET Framework redirects the control to execute the catch or finally blocks whenever an exception has occurred at runtime.
- If we have not handled the Runtime Exception, then the Exception Manager with throw the exception and abnormally terminate the program execution at the line where the exception has occurred.

5. Common Type System (CTS) in .NET Framework:

- The .NET Framework supports many programming languages such as C#, VB.NET, J#, etc. Every programming language has its own data type.

- One programming language data type cannot be understood by other programming languages. But, there can be situations where we need to communicate between two different programming languages.
- For example, we need to write code in the VB.NET language and that code may be called from C# language. In order to ensure smooth communication between these languages,

  the most important thing is that they should have a Common Type System (CTS) which ensures that data types defined in two different languages get compiled to a common data type.
- CLR in .NET Framework will execute all programming language's data types.
- This is possible because CLR has its own data types which are common to all programming languages.
- At the time of compilation, all language-specific data types are converted into CLR's data type.
- This data type system of CLR is common to all .NET Supported Programming languages and this is known as the Common Type System (CTS).



6. CLS (Common Language Specification) in .NET Framework:

- CLS (Common Language Specification) is a part of CLR in the .NET Framework.
- The .NET Framework supports many programming languages such as C#, VB.NET, J#, etc. Every programming language has its own syntactical rules for writing the code which is known as language specification.
- One programming language's syntactical rules (language specification) cannot be understood by other programming languages.
- But, there can be situations where we need to communicate between two different programming languages.
- In order to ensure smooth communication between different .NET Supported Programming Languages, the most important thing is that they should have Common Language Specifications which ensures that language specifications defined in two different languages get compiled into a Common Language Specification.

CLR in .NET Framework will execute all programming language's code. This is possible because CLR has its own language specification (syntactical rules) which are common to all .NET Supported Programming Languages.

- At the time of compilation, every language compiler should follow this language specification of CLR and generate the MSIL code.
- This language specification of CLR is common for all programming languages and this is known as Common Language Specifications (CLS).

❖ Technologies supported by the .NET framework are as follows

1. WinForms:

   Windows Forms is a smart client technology for the .NET Framework, a set of managed libraries that simplify common application tasks such as reading and writing to the file system.

2. ASP.NET:

   ASP.NET is a web framework designed and developed by Microsoft. It is used to develop websites, web applications, and web services. It provides a fantastic integration of HTML, CSS, and JavaScript. It was first released in January 2002.

3. ADO.NET:

   ADO.NET is a module of .Net Framework, which is used to establish a connection between application and data sources. Data sources can be such as SQL Server and XML. ADO .NET consists of classes that can be used to connect, retrieve, insert, and delete data.

4. LINQ (Language Integrated Query):

   It is a query language, introduced in .NET 3.5 framework. It is used to make the query for data sources with C# or Visual Basics programming languages.

QUESTIONS FOR PRACTICE

Q.1) Draw and explain the .net framework architecture.
Q.2) Short note on BCL.
Q.3) Explain CLR & its components.
Q.4) Explain execution process of .net program.
Q.5) Explain MSIL, Native Code and Garbage Collector.

*****

1.0 Introduction to C#

- C# is pronounced as "C-Sharp". It is an object-oriented programming language provided by Microsoft that runs on .Net Framework.
- Anders Hejlsberg is known as the founder of C# language.

With the help of C# programming language, we can develop different types of secured and robust applications:

- Window applications
- Web applications
- Distributed applications
- Web service applications
- Database applications etc.

2.0 C# Features:

C# is object oriented programming language. It provides a lot of **features** that are givenbelow.

1. Simple
2. Object oriented
3. Type safe
4. Interoperability
5. Scalable and Updateable
6. Component oriented
7. Structured programming language
8. Rich Library
9. Fast speed

3.0 Java vs C#:

1)	Java is a high level, robust, secured and object-oriented programming language developed by Oracle.	C# is an object-oriented programming language developed by Microsoft that runs on .Net Framework.
2)	Java programming language is designed to be run on a Java platform, by the help of Java Runtime Environment (JRE).	C# programming language is designed to be run on the Common Language Runtime (CLR).

3)	Java type safety is safe.	C# type safety is unsafe.
7)	Java doesn't support goto statement.	C# supports goto statement.
8)	Java doesn't support structures and unions.	C# supports structures and unions.
9)	Java does not supports for conditional compilation.	C# supports for conditional compilation.

4.0 Structure of C# program:

A C# program consists of the following parts –

- Namespace declaration
- A class
- Class methods
- Class attributes
- A Main method
- Statements and Expressions
- Comments

4.1 Creating Hello World Program:

```
using System;

namespace HelloWorldApplication
{
  class HelloWorld {
    static void Main(string[] args)
{
      /* my first program in C# */
      Console.WriteLine("Hello World");
      Console.ReadKey();
    }
  }
}
```

4.2 Explanation:

- The first line of the program using System; - the using keyword is used to include the System namespace in the program. A program generally has multiple using statements.
- The next line has the namespace declaration. A namespace is a collection of classes. The HelloWorldApplication namespace contains the class HelloWorld.
- The next line has a class declaration, the class HelloWorld contains the dataand method definitions that your program uses. Classes generally contain multiple methods. Methods define the behavior of the class. However, the HelloWorld class has only one method Main.
- The next line defines the Main method, which is the entry point for all C# programs. The Main method states what the class does when executed.
- The next line /*...*/ is ignored by the compiler and it is put to add comments inthe program.
- The Main method specifies its behavior with the statement Console.WriteLine("Hello World");
- WriteLine is a method of the Console class defined in the System namespace. This statement causes the message "Hello, World!" to be displayed on the screen.
- The last line Console.ReadKey(); is for the Users. This makes the program waitfor a key press and it prevents the screen from running and closing quickly when the program is launched from Visual Studio .NET.

5.0 Constants:

- The constants refer to fixed values that the program may not alter during its execution. These fixed values are also called literals.
- Constants can be of any of the basic data types like an integer constant, a floating constant, a character constant, or a string literal.
- There are also enumeration constants as well.

5.1 Defining Constants:

Constants are defined using the **const** keyword.

Syntax:

```
const <data_type> <constant_name> = value;
```

Example:

```
const int a=10;
```

6.0 Variables:

Variables are containers for storing data values.

Declaring (Creating) Variables:

To create a variable, you must specify the type and assign it a value:

Syntax:

Datetype variableName = value;

Example:

int a=12;

7.0 Data type:

A data type specifies the size and type of variable values. It is important to use the correct data type for the corresponding variable; to avoid errors, to save time and memory, but it will also make your code more maintainable and readable. The most common data types are:

Data Type	**Size**	**Descriptio n**
int	4 bytes	-2,147,483,648 to 2,147,483,647
Byte (unsigned)	1 byte	0 to 255
Sbyte (signed)	1 byte	-128 to 127
short	2 byte	-32,768 to 32,767
long	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
ulong	8 bytes	0 to 18,446,744,073,709,551,615

float	4 bytes	Stores fractional numbers. Sufficient for storing 6 to 7decimal digits (suffix 'f')
double	8 bytes	Stores fractional numbers. Sufficient for storing 15decimal digits
bool	1 bit	Stores true or false values
char		Stores a single character/letter, surrounded by single quotes
string		Stores a sequence of characters, surrounded by doublequotes
DateTime	Represent sdate and time	0:00:00am 1/1/01 to 11:59:59pm 12/31/9999
object	Base type of all other types.	The object types can be assigned values of any other types, value types, reference types, predefined or user-defined types.

8.0 Output in C#:

In order to print something on the console, we can use the following two methods
System.Console.WriteLine();
System.Console.Write();

- Here, System is the namespace,
- Console is the class within the namespace System,
- and WriteLine and Write are static methods of the Console class.

8.1 Examples to Print a string on the Console in C#

8.1.1 Program to print a string on the Console window in C#.

```
using System;
```

```
namespace FirstProgram
{
class Program
{
static void Main(string[] args)
{
Console.WriteLine("Hello World!");
}
}
}
```

Output:

```
HELLO WORLD
```

8.2 Difference between WriteLine() and Write() method

- The main difference between WriteLine() and Write() method of Console Classin C# is that the Write() method only prints the string provided to it,
- while the WriteLine() method prints the string and moves to the start of the nextline as well.

8.2.1 Example to understand the use of WriteLine() and Write()

```
using System;
namespace FirstProgram
{
class Program
{
static void Main(string[] args)
{
Console.WriteLine("Prints on ");
Console.WriteLine("New line");
Console.Write("Prints on ");
Console.Write("Same line");
} } }
```

Output:

```
Prints on
New Line
Prints on same line
```

8.3 Variables and Literals using WriteLine() and Write() Method

- The WriteLine() and Write() method of the Console class in C# can also be usedto print variables and literals.

8.3.1 Example Printing Variables and Literals using WriteLine() and Write() Method

```
using System;
namespace FirstProgram
{
class Program
{
static void Main(string[] args)
{
//Printing Variable
int number = 10;
Console.WriteLine(number);
// Printing Literal
Console.WriteLine(50.05);
} } }
```

Output:

```
10
50.05
```

8.4 Printing concatenated string using Formatted String in C#

- A better alternative for printing concatenated strings is using a formatted string instead of the + Operator in C#.
- In the case of formatted strings, we need to use placeholders for variables.
- For example, The following line,

Console.WriteLine(“Number=” + number);

Can be replaced as,

Console.WriteLine(“Number = {0}”, number);

- Here, {0} is the placeholder for the variable number which will be replaced bythe value of the number.
- Since only one variable is used so there is only one placeholder. Multiplevariables can be used in the formatted string. That we will see in our example.

8.4.1 Example to Print Concatenated string using String formatting in C#

- In the below example, {0} is replaced by number1, {1} is replaced by number2and {2} is replaced by sum.
- This approach to printing output is more readable and less error-prone thanusing the + operator.

```
using System;
namespace FirstProgram
{
class Program
{
static void Main(string[] args)
{
int number1 = 15, number2 = 20, sum;
sum = number1 + number2;
Console.WriteLine("{0} + {1} = {2}", number1, number2, sum);
} } }
```

Output:

```
15 + 20 = 35
```

9.0 User Input in C#

- In C#, the simplest method to get input from the user is by using the ReadLine() method of the Console class.
- However, Read() and ReadKey() are also available for getting input from theuser. They are also included in the Console class.
- The most important thing is all these three methods are static methods of the Console class, and hence we can call these methods using the class name

9.1 Example to Get String Input from User in C#:

```
using System;
namespace FirstProgram
{
   class Program
   {
      static void Main(string[] args)
```

```
    {
      string str;
      Console.Write("Enter a string - ");
      str = Console.ReadLine();
      Console.WriteLine("You entered " + str);
    } } }
```

9.2 Difference between ReadLine(), Read() and ReadKey() methods

1. ReadLine(): The ReadLine() method of Console class in C# reads the next line of input from the standard input stream. It returns the same string.
2. Read(): The Read() method of Console class in C# reads the next character from the standard input stream. It returns the ASCII value of the character.
3. ReadKey(): The ReadKey() method of the Console class in C# obtains the next key pressed by the user. This method is usually used to hold the screen until the userpress a key.

9.2.1 Example 1 of Read():

```
using System;
namespace ConsoleApp1
{
  class Program
  {
    static void Main(string[] args)
    {
      int a;
      a = Console.Read();
      Console.Write("value of a= " + a);
    } }
}
```

Output:

```
a
value of a= 97
```

9.2.2 Example 2 of Read():

```
using System;
```

```
namespace ConsoleApp1
{
   class Program
   {
      static void Main(string[] args)
      {
         char ch;
         ch = Convert.ToChar(Console.Read());
         Console.Write("value of a= " + ch);
      } }
}
```

Output:

```
ABC
value of a= A
```

9.3 Reading Integer and floating Numbers (Numeric Values)

- In C#, it is very easy to read a character or string. All we need to do is call the corresponding methods as required like Read, ReadKey, and ReadLine.
- But it is not that straightforward while read the numeric values. Here, we willuse the same ReadLine() method we used for getting string values.
- But since the ReadLine() method receives the input as a string, we need to typecast it into an integer or floating-point type as per our requirement.
- The simplest approach for converting user input to integer or floating-pointtype is by using the methods of the Convert class.

9.3.1 Example to Read Numeric Values from User Using Convert class.

```
using System;
namespace FirstProgram
{
   class Program
   {
      static void Main(string[] args)
      {
          int Val;
         double doubleVal;
```

```
        Console.Write("Enter integer value: ");
        // Converts to integer type
        Val = Convert.ToInt32(Console.ReadLine());
        Console.WriteLine("You entered {0}", Val);

        Console.Write("Enter double value: ");
        // Converts to double type
        doubleVal = Convert.ToDouble(Console.ReadLine());
        Console.WriteLine("You entered {0}", doubleVal);
    } } }
```

Note: The ToInt32() and ToDouble() method of the Convert class converts the string input to integer and double type respectively. Similarly, you can convert the input to other types.

10.0 Operators in C#

- Operators in C# are symbols that are used to perform operations on operands.
- For example, consider the expression **2 + 3 = 5**, here **2 and 3 are operands**,and **+ and = are called operators**.
- So, the Operators in C# are used to manipulate the variables and values in a program.

10.1 Types of Operators in C#:

The Operators are classified based on the type of operations they perform onoperands in C# language. They are as follows:

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Assignment Operators
5. Increment & Decrement Operator
6. Ternary Operator or Conditional Operator

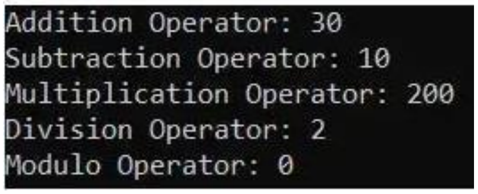
1. Arithmetic Operators

- The Arithmetic Operators in C# are used to perform arithmetic/mathematical operations like addition, subtraction, multiplication, division, etc. on operands. The following Operators are falling into this category.
  +,-,*,/,%

10.1.1 Programs for Arithmetic:

```
using System;
namespace OperatorsDemo
{
   class Program
   {
      static void Main(string[] args)
      {
         int Result;
         int Num1 = 20, Num2 = 10;

         // Addition Operation
         Result = (Num1 + Num2);
         Console.WriteLine($"Addition Operator: {Result}" );

         // Subtraction Operation
         Result = (Num1 - Num2);
         Console.WriteLine($"Subtraction Operator: {Result}");

         // Multiplication Operation
         Result = (Num1 * Num2);
         Console.WriteLine($"Multiplication Operator: {Result}");

         // Division Operation
         Result = (Num1 / Num2);
         Console.WriteLine($"Division Operator: {Result}");

         // Modulo Operation
         Result = (Num1 % Num2);
         Console.WriteLine($"Modulo Operator: {Result}");
         Console.ReadKey();
      } }  }
```

Output:



```
Addition Operator: 30
Subtraction Operator: 10
Multiplication Operator: 200
Division Operator: 2
Modulo Operator: 0
```

2. Relational Operators

- The Relational Operators in C# are also known as Comparison Operators.
- It determines the relationship between two operands and returns the Boolean results, i.e. true or false after the comparison.
- The Different Types of Relational Operators supported by C# are as follows.

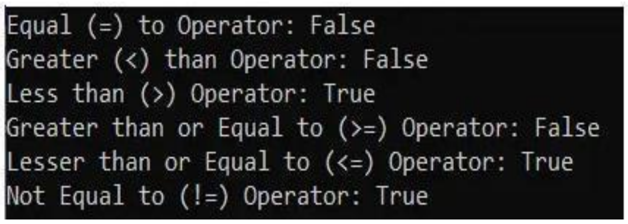
  < , <=, >, >=, ==, !=

10.1.2 Program for relational operator:

```
using System;
namespace OperatorsDemo
{
   class Program
   {
      static void Main(string[] args)
      {
         bool Result;
         int Num1 = 5, Num2 = 10;

         // Equal to Operator
         Result = (Num1 == Num2);
         Console.WriteLine("Equal (=) to Operator: " + Result);

         // Greater than Operator
         Result = (Num1 > Num2);
         Console.WriteLine("Greater (<) than Operator: " + Result);

         // Less than Operator
         Result = (Num1 < Num2);
         Console.WriteLine("Less than (>) Operator: " + Result);

         // Greater than Equal to Operator
```

```
            Result = (Num1 >= Num2);
            Console.WriteLine("Greater than or Equal to (>=) Operator: " + Result);

            // Less than Equal to Operator
            Result = (Num1 <= Num2);
            Console.WriteLine("Lesser than or Equal to (<=) Operator: " + Result);

            // Not Equal To Operator
            Result = (Num1 != Num2);
            Console.WriteLine("Not Equal to (!=) Operator: " + Result);
            Console.ReadKey();
        } } }
```

Output:



```
Equal (=) to Operator: False
Greater (<) than Operator: False
Less than (>) Operator: True
Greater than or Equal to (>=) Operator: False
Lesser than or Equal to (<=) Operator: True
Not Equal to (!=) Operator: True
```

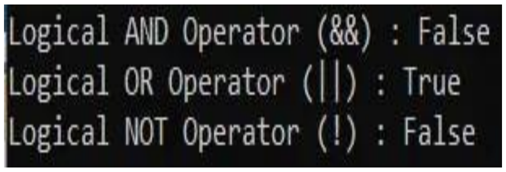
3. Logical Operators

- The Logical Operators are mainly used in conditional statements and loops for evaluating a condition. These operators are going to work with boolean expressions.
- The different types of Logical Operators supported in C# are as follows:
- &&, | |, !

10.1.3 Program for logical operator:

```
using System;
namespace OperatorsDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            bool x = true, y = false, z;

            //Logical AND operator
            z = x && y;
            Console.WriteLine("Logical AND Operator (&&) : " + z);

            //Logical OR operator
            z = x || y;
            Console.WriteLine("Logical OR Operator (||) : " + z);

            //Logical NOT operator
            z = !x;
            Console.WriteLine("Logical NOT Operator (!) : " + z);
            Console.ReadKey();
        }}}
```

Output:



```
Logical AND Operator (&&) : False
Logical OR Operator (||) : True
Logical NOT Operator (!) : False
```

4. Assignment Operator (=)

- The Assignment Operators in C# are used to assign a value to a variable.
- The left-hand side operand of the assignment operator is a variable and the right-hand side operand of the assignment operator can be a value or an expression that must return some value and that value is going to assign to the left-hand side variable.

10.1.4 Simple Assignment (=):

This operator is used to assign the value of the right-hand side operand to theleft-hand side operand i.e. to a variable.

For example:

int a=10;

int b=20;

char ch = ‘a’;

a=a+4; //(a=10+4)

b=b-4; //(b=20-4)

5. Shorthand assignment Operator:

- Following operators comes under shorthand assignment operator

- +=, -=, *=, /=, %=

i. Add Assignment (+=):

This operator is the combination of + and = operators. It is used to add the left- hand side operand value with the right-hand side operand value and then assignthe result to the left-hand side variable.

For example:

int a=5;

int b=6;

a += b; //a=a+b; That means (a += b) can be written as (a = a + b)

ii.Subtract Assignment (-=):

This operator is the combination of – and = operators. It is used to subtract theright-hand side operand value from the left-hand side operand value and then assign the result to the left-hand side variable.

For example:

int a=10;

int b=5;

a -= b; //a=a-b; That means (a -= b) can be written as (a = a – b)

Multiply Assignment (*=):

This operator is the combination of * and = operators. It is used to multiply theleft-hand side operand value with the right-hand side operand value and then assign the result to the left-hand side variable.

For example:

int a=10;

int b=5;

a *= b; //a=a*b; That means (a *= b) can be written as (a = a * b)

Division Assignment (/=):

This operator is the combination of / and = operators. It is used to divide the left-hand side operand value with the right-hand side operand value and then assign the result to the left-hand side variable.

For example:
int a=10;
int b=5;
a /= b; //a=a/b; That means (a /= b) can be written as (a = a / b)

Modulus Assignment (%=):
This operator is the combination of % and = operators. It is used to divide the left-hand side operand value with the right-hand side operand value and then assigns the remainder of this division to the left-hand side variable.

For example:
int a=10;
int b=5;
a %= b; //a=a%b; That means (a %= b) can be written as (a = a % b)

6. Increment Operator (++)
The Increment Operator (++) is a unary operator. It operates on a single operandonly. Again, it is classified into two types:

1. Post-Increment Operator
2. Pre-Increment Operator

1. Post Increment Operators:
The Post Increment Operators are the operators that are used as a suffix to its variable. It is placed after the variable. For example, a++ will also increase the valueof the variable a by 1.

Syntax: Variable++;
Example: x++;
2. Pre-Increment Operators:
The Pre-Increment Operators are the operators which are used as a prefix to its variable. It is placed before the variable. For example, ++a will increase the value ofthe variable a by 1.

Syntax: ++Variable;
Example: ++x;

7. Decrement Operators:
The Decrement Operator (–) is a unary operator. It takes one value at a time. It isagain classified into two types. They are as follows:

1. Post Decrement Operator
2. Pre-Decrement Operator

1. Post Decrement Operators:
The Post Decrement Operators are the operators that are used as a suffix to its variable. It is placed after the variable. For example, a– will also decrease the valueof the variable a by 1.
Syntax: Variable--;
Example: x--;
2. Pre-Decrement Operators:
The Pre-Decrement Operators are the operators that are a prefix to its variable. It is placed before the variable. For example, –a will decrease the value of the variablea by 1.
Syntax:--Variable;
Example: --x;

8. Ternary Operator

- The Ternary Operator in C# is also known as the Conditional Operator (**?:**).
- It is actually the shorthand of the if-else statement. It is called ternary becauseit has three operands or arguments.
- The first argument is a comparison argument, the second is the result of a true comparison, and the third is the result of a false comparison.

Syntax: Condition? first_expression : second_expression;
The above statement means that first, we need to evaluate the condition. If the condition is true the first_expression is executed and becomes the result and if the condition is false, the second_expression is executed and becomes the result.

10.1.5 Program for ternary operator:

```
using System;
```

```
namespace OperatorsDemo
{
   class Program
   {
      static void Main(string[] args)
      {
         int a = 20, b = 10, res;
         res = ((a > b) ?a : b);
         Console.WriteLine("Result = " + res);
         Console.ReadKey();
      } } }
```

Output:

```
Result = 20
```

11.0 Control/Conditional Statement:

A statement that can be executed based on a condition is known as a "Conditional Statement". The statement is often a block of code.

C#, the control flow statements are divided into the following three categories:

1. Selection Statements or Branching Statements: (Examples: if-else, switchcase, nested if-else, if-else ladder)
2. Iteration Statements or Looping Statements: (Examples: while loop, do-whileloop, for-loop, and foreach loop)
3. Jumping Statements: (Examples: break, continue, return, goto)

11.1 Conditional Branching Statements:

This statement allows you to branch your code depending on whether or not a certain condition is met.

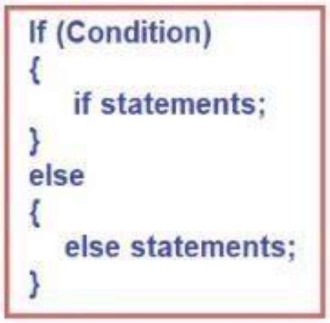
In C# are the following are conditional branching statements:

1. If-else
2. Nested if
3. Else if ladder
4. Switch

11.1.1 If-else:

- The If-Else block in C# Language is used to provide some optional information whenever the given condition is FALSE in the if block.
- That means if the condition is true, then the if block statements will be executed, and if the condition is false, then the else block statement will execute.

Syntax:



```
If (Condition)
{
    if statements;
}
else
{
   else statements;
}
```

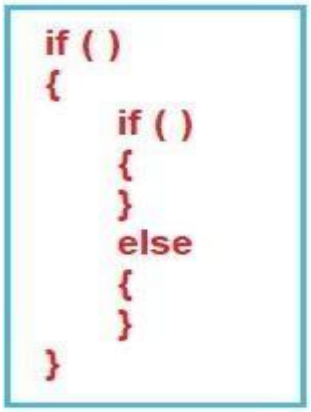
Example:

```
using System;
namespace ControlFlowDemo
{
   class Program
   {
      static void Main(string[] args)
      {
         Console.WriteLine("Enter a Number: ");
         int number = Convert.ToInt32(Console.ReadLine());
         if (number % 2 == 0)
         {
            Console.WriteLine($"{number} is an Even Number");
         }
         else
         {
            Console.WriteLine($"{number} is an Odd Number");
         }
         Console.ReadKey();
      } } }
```

11.1.2. Nested If-Else:

- When an if-else statement is present inside the body of another if or else thenthis is called nested if-else.
- Nested IF-ELSE statements are used when we want to check for a condition onlywhen the previous dependent condition is true or false.

Syntax



```
if ( )
{
    if ( )
    {
    }
    else
    {
    }
}
```

Program:

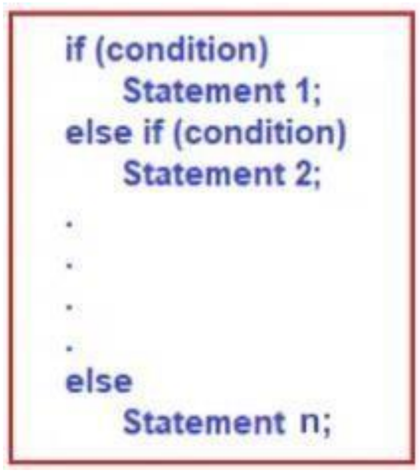
```
using System;
namespace ControlFlowDemo
{
   class Program
   {
      static void Main(string[] args)
      {
         int a = 15, b = 25, c = 10;
         int LargestNumber = 0;

         if (a > b)
         {
            Console.WriteLine($"Outer IF Block");
            if (a > c)
            {
               Console.WriteLine($"Outer IF - Inner IF Block");
               LargestNumber = a;
            }
            else
```

```
        {
            Console.WriteLine($"Outer IF - Inner ELSE Block");
            LargestNumber = c;
        }
    }
    else
    {
        Console.WriteLine($"Outer ELSE Block");
        if (b > c)
        {
            Console.WriteLine($"Outer ELSE - Inner IF Block");
            LargestNumber = b;
        }
        else
        {
            Console.WriteLine($"Outer ELSE - Inner ELSE Block");
            LargestNumber = c;
        }
    }
    Console.WriteLine($"The Largest Number is: {LargestNumber}");
    Console.ReadKey();
} } }
```

11.1.3. if-else if ladder statements:

- In Ladder if-else statements one of the statements will be executed depending upon the truth or false of the conditions.
- If the condition1 is true then Statement 1 will be executed, and if condition2 is true then statement 2 will be executed, and so on.
- But if all conditions are false, then the last statement i.e. else block statement will be executed.
- The C# if-else statements are executed from top to bottom. As soon as one of the conditions controlling the if is true, the statement associated with that if block isgoing to be executed, and the rest of the C# else-if ladder is bypassed.
- If none of the conditions are true, then the final else statement will be executed.

Syntax



```
if (condition)
    Statement 1;
else if (condition)
    Statement 2;
.
.
.
.
else
    Statement n;
```

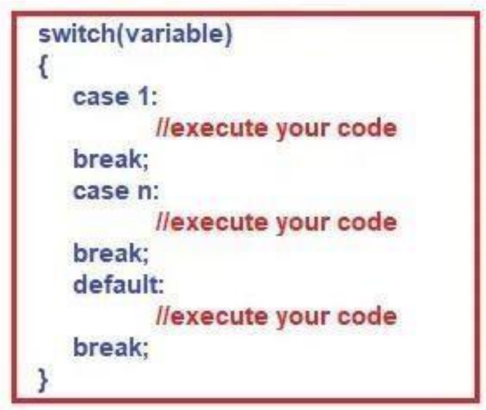
Programs:

```
using System;
namespace ControlFlowDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            int i = 20;
            if (i == 10)
            {
                Console.WriteLine("i is 10");
            }
            else if (i == 15)
            {
                Console.WriteLine("i is 15");
            }
            else if (i == 20)
            {
                Console.WriteLine("i is 20");
            }
            else
            {
                Console.WriteLine("i is not present");
            }
        Console.ReadKey();
    } } }
```

11.14. Switch statement

- Switch case statements in C# are a substitute for long if else statements that compare a variable or expression to several values.
- The switch statement is a multi-way branching statement which means itprovides an easy way to switch the execution to different parts of code based on the value of the expression.
- The switch expression is of integer type such as int, byte, or short, or of an enumeration type, or of character type, or of string type. The expression is checked for different cases and the match case will be executed.

Syntax:



```
switch(variable)
{
    case 1:
            //execute your code
    break;
    case n:
            //execute your code
    break;
    default:
            //execute your code
    break;
}
```

Program 1:

```
using System;
namespace ControlFlowDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            int x = 2;
            switch (x)
            {
                case 1:
                    Console.WriteLine("Choice is 1");
                break;
            case 2:
                Console.WriteLine("Choice is 2");
                break;
            case 3:
                Console.WriteLine("Choice is 3");
                break;
            default:
                Console.WriteLine("Choice other than 1, 2 and 3");
                break;
        }
        Console.ReadKey();
    } } }
```

Program 2:

```
using System;
namespace ControlFlowDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            string str = "C#";
            switch (str)
            {
                case "C#":
                case "Java":
                case "C":
                    Console.WriteLine("It’s a Programming Langauge");
                    break;

                case "MSSQL":
                case "MySQL":
                case "Oracle":
                    Console.WriteLine("It’s a Database");
                    break;
            case "MVC":
            case "WEB API":
               Console.WriteLine("It’s a Framework");
               break;

            default:
               Console.WriteLine("Invalid Input");
               break;
         }
         Console.ReadKey();
      }
   }
}
```

Output:

```
It’s a Programming Language
```

12.0 Loops in c#:

Loops are also called repeating statements or iterative statements. Loops play an important role in programming. The Looping Statements are also called Iteration Statements.

12.1 Types of Loops:

1. For loop
2. For Each Loop
3. While loop
4. Do while loop

Loops are mainly divided into two categories:

1. **Entry Controlled Loops:** The loops in which the condition to be tested is present at beginning of the loop body are known as Entry Controlled Loops.Examples of Entry Controlled Loops are while loop and for loop.
2. **Exit Controlled Loops:** The loops in which the testing condition is present at the end of the loop body are termed Exit Controlled Loops. An example of Exit Controlled Loop is the do-while loop. In Exit Controlled Loops, the loop body will be evaluated at least one time as the testing condition is present atthe end of the loop body.

12.1.1 While loop:

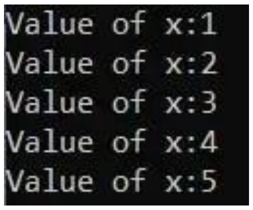
Syntax:

```
Initialization;
While(Codition)
{
Statement;
Incr/decr;
}
```

Program:

```
using System;
namespace ControlFlowDemo
{
   class Program
   {
      static void Main(string[] args)
      {
         int x = 1;
         while (x <= 5)
         {
            Console.WriteLine("Value of x:" + x);
            x++;
         }
         Console.ReadKey();
      } } }
```

Output:



```
Value of x:1
Value of x:2
Value of x:3
Value of x:4
Value of x:5
```

11.1.2. Do while loop:

- The do-while loop is a post-tested loop or exit-controlled loop i.e. first it will execute the loop body and then it will be going to test the condition.
- That means we need to use the do-while loop where we need to execute theloop body at least once.

Syntax:

```
Initialization;
Do
{
Statement;
Incr/decr;
} While(Codition);
```

Program:

```
using System;
namespace ControlFlowDemo
{
   class Program
   {
      static void Main(string[] args)
      {
         int number = 1;
         do
         {
            Console.Write($"{number} ");
            number++;
         } while (number <= 5);

         Console.ReadKey();
      } } }
```

11.1.3. For Loop:

For loop is one of the most commonly used loops in the C# language. If we know the number of times, we want to execute some set of statements or instructions,then we should use for loop.

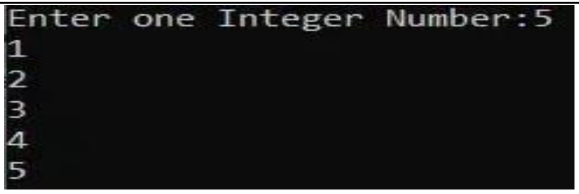
Syntax:

```
For(initialization; condition; incr/decr)
{
Statements;
}
```

Program:

```
using System;
namespace ControlFlowDemo
{
   class Program
   {
      static void Main(string[] args)
      {
         Console.Write("Enter one Integer Number:");
         int number = Convert.ToInt32(Console.ReadLine());
         for (int counter = 1; counter <= number; counter++)
         {
            Console.WriteLine(counter);
         }
         Console.ReadKey();
      } } }
```

Output:

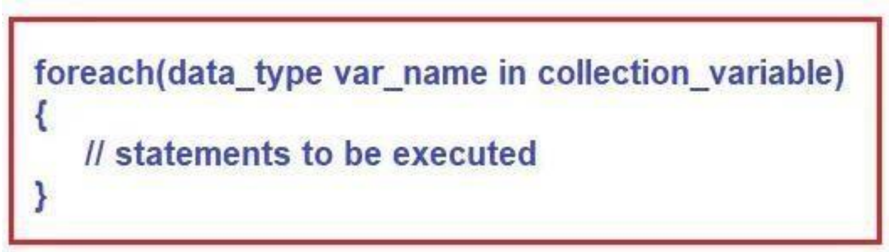


```
Enter one Integer Number:5
1
2
3
4
5
```

11.1.4. Foreach Loop:

- The Foreach Loop in C# is a different kind of loop that doesn't include initialization, termination, and increment/decrement characteristics. It uses thecollection to take values one by one and then processes them.
- The foreach loop in C# is used to iterate over the elements of a collection. Here, the collection may be an array or a list, etc. As per the name i.e. foreach, it executes the loop body for each element present in the array or collection.

Syntax:

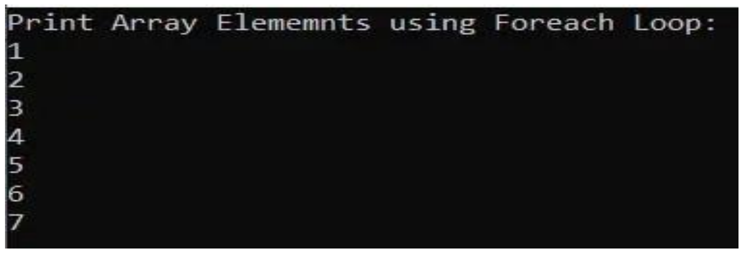


```
foreach(data_type var_name in collection_variable)
{
    // statements to be executed
}
```

Program:

```
using System;
namespace ForeachLoopDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            // creating an array of integer type
            int[] IntArray = new int[] { 1, 2, 3, 4, 5, 6, 7 };

            Console.WriteLine("Print Array Elememnts using Foreach Loop:");
            // The foreach loop will run till the last element of the arrayforeach
            (int item in IntArray)
            {
                Console.WriteLine(item);
            }
            Console.ReadKey();
        } } }
```

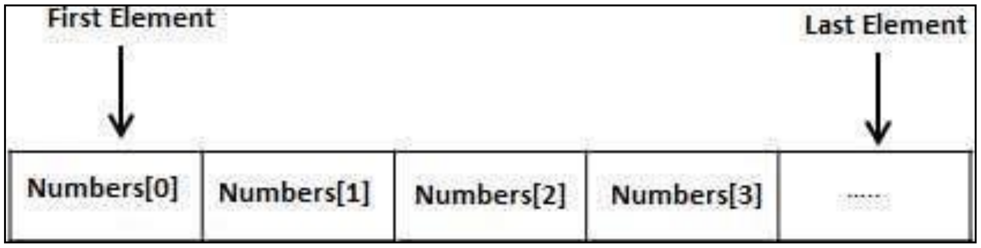
Output:



```
Print Array Elememnts using Foreach Loop:
1
2
3
4
5
6
7
```

12.0 Array & its function:

- An array stores a fixed-size sequential collection of elements of the same type.An array is used to store a collection of data.
- Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.
- All arrays consist of contiguous memory locations.



12.1 Array Types

There are 3 types of arrays in C# programming:

1. Single Dimensional Array
2. Multidimensional Array
3. Jagged Array

12.1.1 Single Dimensional Array

- To create single dimensional array, you need to use square brackets [] afterthe type.

12.1.1.1 Syntax to declare an array:

Datatype [] variable = new datatype [size];

Example:

int[] arr = new int[5];

You cannot place square brackets after the identifier.

int arr[] = new int[5]; //compile time error

12.1.1.2 Declaration and Initialization at same time

- There are 3 ways to initialize array at the time of declaration.

1. int[] arr = new int[5]{ 10, 20, 30, 40, 50 };
2. int[] arr = new int[]{ 10, 20, 30, 40, 50 };
3. int[] arr = { 10, 20, 30, 40, 50 };

12.1.1.3 Example of array where we are declaring and initializing array at thesame time.

```
using System;
public class ArrayExample
{
   public static void Main(string[] args)
   {
      int[] arr = { 10, 20, 30, 40, 50 }; //Declaration and Initialization of array

      //traversing array
      for (int i = 0; i < arr.Length; i++)
      {
         Console.WriteLine(arr[i]);
      }
   }
}
```

Output:

```
10
20
30
40
50
```

12.1.1.4 Traversal using foreach loop

```
using System;
public class ArrayExample
{
   public static void Main(string[] args)
   {
      int[] arr = { 10, 20, 30, 40, 50 };//creating and initializing array
```

```
        //traversing array
        foreach (int i in arr)
        {
           Console.WriteLine(i);
        }
    }
}
```

Output:

```
10
20
30
40
50
```

12.1.1.5 Taking input from user in 1-d array.

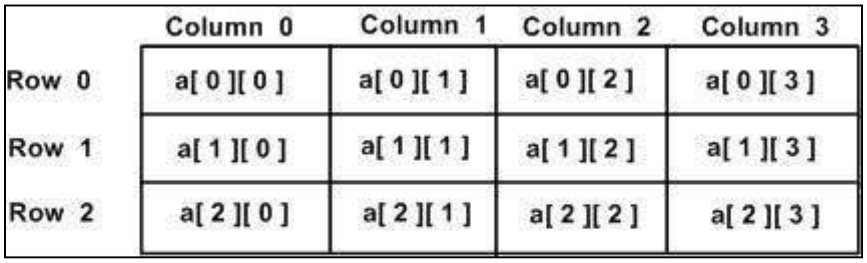
```
using System;
public class ArrayExample
{
    public static void Main(string[] args)
    {
        int[] arr = new int[4];

        Console.WriteLine("Enter values:");
        for (int i = 0; i < arr.Length; i++)
        {
           arr[i] = Convert.ToInt32(Console.ReadLine());
        }

        Console.WriteLine("Values are:");
        for (int i = 0; i < arr.Length; i++)
        {
           Console.WriteLine(arr[i]);
        }  } }
```

Output:

```
Enter values:
1
3
4
5
Values are:
1
3
4
5
```

12.1.2 Multi Dimensional Array

- The multidimensional array is also known as rectangular arrays in C#.
- The data is stored in tabular form (row * column) which is also known as matrix.
- To create multidimensional array, we need to use comma inside the square brackets.
- A 2-dimensional array can be thought of as a table, which has x number of rowsand y number of columns. Following is a 2-dimensional array, which contains 3 rows and 4 columns –



	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

<u>Example to create multi dimensional array:</u>

```
int[,] arr=new int[3,3];//declaration of 2D array
```

<u>Initializing Two-Dimensional Arrays</u>

```
int [,] a = new int [3,4] { {0, 1, 2, 3} , {4, 5, 6, 7} , {8, 9, 10, 11} };
```

<u>Multidimensional Array Example</u>

```
using System;
```

```
public class MultiArrayExample
{
    public static void Main(string[] args)
    {
        int[,] arr=new int[3,3];//declaration of 2D array
        arr[0,1]=10;//initialization
        arr[1,2]=20;
        arr[2,0]=30;

        //traversal
        for(int i=0;i<3;i++){
            for(int j=0;j<3;j++){
                Console.Write(arr[i,j]+" ");
            }
            Console.WriteLine();//new line at each row
        }
    }
}
```

Output:

```
0  10   0
0  0    20
30 0    0
```

12.1.3 Jagged Array

- Jagged array is also known as "array of arrays" because its elements are arrays.
- The element size of jagged array can be different for each row.
- In Jagged array we can specify different column size for each row.

Declaration of Jagged array

int[][] arr = new int[2][]; //2 is size of row

Initialization of Jagged array

arr[0] = new int[4]; //1st row 4 columns

arr[1] = new int[6]; //2nd row 6 columns

Initialization and filling elements in Jagged array

```
arr[0] = new int[4] { 11, 21, 56, 78 };
arr[1] = new int[6] { 42, 61, 37, 41, 59, 63 };
```

Here, size of elements in jagged array is optional. So, you can write above code asgiven below:

```
arr[0] = new int[] { 11, 21, 56, 78 };
arr[1] = new int[] { 42, 61, 37, 41, 59, 63 };
```

Jagged Array Example 1

```
public class JaggedArrayTest
{
    public static void Main()
    {
        int[][] arr = new int[2][];// Declare the array

        arr[0] = new int[] { 11, 21, 56, 78 };// Initialize the array
        arr[1] = new int[] { 42, 61, 37, 41, 59, 63 };

        // Traverse array elements
        for (int i = 0; i < arr.Length; i++)
        {
            for (int j = 0; j < arr[i].Length; j++)
            {
                System.Console.Write(arr[i][j]+" ");
            }
            System.Console.WriteLine();
        } } }
```

Output:

```
11 21 56 78
42 61 37 41 59 63
```

Initialization of Jagged array upon Declaration

```
  int[][] arr = new int[3][]
 {
```

```
new int[] { 11, 21, 56, 78 },
new int[] { 2, 5, 6, 7, 98, 5 },
new int[] { 2, 5 }
};
```

Jagged Array Example 2

```
public class JaggedArrayTest
{
    public static void Main()
    {
        int[][] arr = new int[3][]{
        new int[] { 11, 21, 56, 78 },
        new int[] { 2, 5, 6, 7, 98, 5 },
        new int[] { 2, 5 }
        };

        // Traverse array elements
        for (int i = 0; i < arr.Length; i++)
        {
            for (int j = 0; j < arr[i].Length; j++)
            {
                System.Console.Write(arr[i][j]+" ");
            }
            System.Console.WriteLine();
        }
    }
}
```

Output:

```
11 21 56 78
2 5 6 7 98 5
2 5
```

13.0 Array properties/methods

The Array class is the base class for all the arrays in C#. It is defined in the System namespace. The Array class provides various properties and methods to work with arrays.

13.1 MOST COMMON PROPERTIES OF ARRAY CLASS

Properties	**Explanation**	**Example**
Length	Returns the length of array. Returns integer value.	int i = arr1.Length;
Rank	Returns total number of items in all the dimension. Returns integer value.	int i = arr1.Rank;

13.2 MOST COMMON METHODS OF ARRAY CLASS

1. Array.Copy()
2. Array.Sort()
3. Array.IndexOf()
4. Array.Reverse()

1. Array.Copy()

- Copies a range of elements from an Array starting at the first element and pastesthem into another Array starting at the first element.

  Syntax:

  Array.Copy(Source_Array, Dest_Array, Length);

  (Length Specifies total number of elements to be copied from source array)

  Program:

```
using System;
public class ArrayExample
{
   public static void Main(string[] args)
   {
        int[] a = new int[4] { 12, 32, 44, 55 };
        int[] b = new int[4];

        Array.Copy(a, b, 4);

        foreach(int i in b)
        {
           Console.Write(i + " ");
        }
    }
}
```

Output:

```
12 32 44 55
```

2. Array.Sort()

- It sorts the array elements in ascending order
- It sorts all types of data

Syntax:

Array.Sort (Array);

Program:

```
using System;
public class ArrayExample
{
    public static void Main(string[] args)
    {
        int[] a = new int[4] { 122, 32, 5, 55 };


        Array.Sort(a);

        foreach(int i in a)
        {
           Console.Write(i + " ");
        } } }
```

Output: 5 32 55 122

4. Array.IndexOf()

- It returns index number of the passed element.
- If element not found in array,then it returns -1.

Syntax:
Array.Indexof(Array, Value);

Program:

```
using System;
public class ArrayExample
{
   public static void Main(string[] args)
   {
      int[] a = new int[4] { 122, 32, 5, 55 };

      int i=Array.IndexOf(a, 5);

      Console.WriteLine("i=" + i);

      int j = Array.IndexOf(a, 99);

      Console.WriteLine("j="+j);
   } }
```

Output:

```
i=2
j=-1
```

14.0 Arraylist

- In C#, an ArrayList stores elements of multiple data types whose size can be changed dynamically.
- To create ArrayList in C#, we need to use the **System.Collections** namespace.

Create an ArrayList

ArrayList myList = new ArrayList();

14.1 Basic Operations on ArrayList

In C#, we can perform different operations on arraylists. We will look at some commonly used arraylist operations in this tutorial:

1. Add Elements
2. Access Elements
3. Change Elements
4. Remove Elements

14.1.1 Add Elements in ArrayList

C# provides a method Add() using which we can add elements in ArrayList. For example,

```
using System;
using System.Collections;
class Program
{
   public static void Main()
   {
      // create an ArrayList
      ArrayList student = new ArrayList();

      // add elements to ArrayList
      student.Add("Tina");
      student.Add(5);
   }
}
```

14.1.2 Access ArrayList

- We use indexes to access elements in ArrayList. The indexing starts from **0**.
- myList.Count gives the number of elements in myList

Program:

```
using System;
using System.Collections;
class Program
{
   public static void Main()
   {
      // create an ArrayList containing 3 elements
      ArrayList myList = new ArrayList();

      myList.Add("Science");
      myList.Add(true);
      myList.Add(5);

      // display every element of myList
      foreach(object j in myList)
      {
         Console.WriteLine(j);
      } } }
```

14.1.3 Change ArrayList

We can change the value of elements in ArrayList as:

```
using System;
using System.Collections;
class Program
{
   public static void Main()
   {
      // create an ArrayList
      ArrayList myList = new ArrayList();

      myList.Add("Harry");
      myList.Add("Miller");

      Console.WriteLine("Original Second element: " + myList[1]);

      // change the value of second element
      myList[1] = "Styles";
```

```
    Console.WriteLine("Updated second element: " + myList[1]);
} }
```

Output:

```
Original Second element: Miller
Updated second element: Styles
```

14.1.4 Remove ArrayList Elements

- C# provides methods like Remove(), RemoveAt() to remove elements from ArrayList.

1. **Remove():**

Remove() methods accepts element to be removed.

Program:

```
using System;
using System.Collections;
class Program
{
  public static void Main()
  {
    // create an ArrayList
    ArrayList myList = new ArrayList();
    myList.Add("Jack");
    myList.Add(4);
    myList.Add("Jimmy");
    myList.Remove("Jack");

    // iterate through myList after removing "Jack"
    for (int i = 0; i < myList.Count; i++)
    {
      Console.WriteLine(myList[i]);
    }
    Console.ReadKey();
  }
}
```

Output:

```
4
Jimmy
```

2. RemoveAt():

RemoveAt() methods accepts index number of the element to be removed.

Program:

```
using System;
using System.Collections;

class Program
{
    public static void Main()
    {
        // create an ArrayList
        ArrayList myList = new ArrayList();
        myList.Add("Jack");
        myList.Add(4);
        myList.Add("Jimmy");

        myList.RemoveAt(1);

        for (int i = 0; i < myList.Count; i++)
        {
            Console.WriteLine(myList[i]);
        }
        Console.ReadKey();
    }
}
```

Output:

```
Jack
Jimmy
```

15.0 String functions

String Functions	**Definitions**
Contains()	The C# Contains method checks whether specified character or string is exists or not in the string value.
EndsWith()	This EndsWith Method checks whether specified characteris the last character of string or not.
Equals()	The Equals Method in C# compares two string and returns Boolean value as output.
IndexOf()	Returns the index position of first occurrence of specified character.
ToLower()	Converts String into lower case based on rules of the current culture.
ToUpper()	Converts String into Upper case based on rules of the current culture.
Insert()	Insert the string or character in the string at the specified position.
LastIndexOf()	Returns the index position of last occurrence of specified character.
Length	It is a string property that returns length of string.
Remove()	This method deletes all the characters from beginning to specified index position.

Replace()	This method replaces the character.
StartsWith()	It checks whether the first character of string is same as specified character.
Substring()	This method returns substring.
Trim()	It removes extra whitespaces from beginning and ending of string.

Program:

```
using System;
namespace string_function
{
    class Program
    {
        static void Main(string[] args)
        {
            string firstname;
            string lastname;

            firstname = "Steven Clark";
            lastname = "Clark";

             //Check whether specified value exists or not in string
             Console.WriteLine(firstname.Contains("ven"));

//Check whether specified value is the last character of string
              Console.WriteLine(firstname.EndsWith("n"));


Console.WriteLine(firstname.Equals(lastname));
              //Compare two string and returns true and false

            Console.WriteLine(firstname.IndexOf("e"));
            //Returns the first index position of specified value the first index positionof
specified value

             Console.WriteLine(firstname.ToLower());
            //Covert string into lower case

            Console.WriteLine(firstname.ToUpper());
            //Convert string into Upper case

            Console.WriteLine(firstname.Insert(0, "Hello"));
//Insert substring into string

            Console.WriteLine(firstname.LastIndexOf("e"));
//Returns the last index position of specified value

            Console.WriteLine(firstname.Length);
            //Returns the Length of String

            Console.WriteLine(firstname.Remove(5));
            //Deletes all the characters from begining to specified index.

            Console.WriteLine(firstname.Replace('e', 'i'));
// Replace the character

            Console.WriteLine(firstname.StartsWith("S"));
 //Check wheter first character of string is same as specified value

            Console.WriteLine(firstname.Substring(2, 5));
            //Returns substring

            Console.WriteLine(firstname.Trim());
            //It removes starting and ending white spaces from string.
        } } }
```

Output:

```
True
False
False
```

```
2
steven clark
STEVEN CLARK
HelloSteven Clark4
12
Steve Stivin
ClarkTrue
even
Steven Clark
```

16.0 Maths Functions

The Math class comes under System namespace, and it is used for trigonometric, logarithmic, and other common mathematical functions.

Method	Description
Abs()	Returns the absolute value of a specified number.
Ceiling()	Returns the smallest integral value greater than or equal to the specified number.
Floor()	Returns the largest integral value less than or equal to the specified number.
Max()	Returns the larger of two specified numbers.
Min()	Returns the smaller of two numbers.
Pow()	Returns a specified number raised to the specified power.

Method	Description
Round()	Rounds a value to the nearest integer or to the specified number of fractional digits.
Sign()	Returns an integer that indicates the sign of a number.
Sqrt()	Returns the square root of a specified number.
Truncate()	Calculates the integral part of a number.

Program:

```
using System;
public class GFG
{
        static public void Main()
        {

                // using Abs() Method
                Console.WriteLine("Absolute Value= " + Math.Abs(-43));

                // using Floor() Method
                Console.WriteLine("Floor value of 123.123: " + Math.Floor(123.623));

                // using Ceiling() Method
        Console.WriteLine("Ceiling value of 123.12: " + Math.Ceiling(123.12));

                // using Sqrt() Method
        Console.WriteLine("Square Root of 81: " + Math.Sqrt(81));

                // using Round() Method
        Console.WriteLine("Round value of 14.6534: " + Math.Round(14.6534));

                // using Min() Method
                Console.WriteLine("Minimum= " + Math.Min(32,65));
                // using Max() Method
                Console.WriteLine("Maximum= " + Math.Min(102, 265));

                //          using          Pow()          Method
                Console.WriteLine("Power= " + Math.Pow(3,2));
        }
}
```

Output:

```
Absolute Value= 43
Floor value of 123.123: 123
Ceiling value of 123.12: 124
Square Root of 81: 9
Round value of 14.6534: 15
Minimum= 32
Maximum= 102
Power= 9
```

17.0 Methods

A method is a group of statements that together perform a task. Every C# programhas at least one class with a method named Main.

To use a method, you need to −

- Define the method
- Call the method

17.1 Defining Methods in C#

When you define a method, you basically declare the elements of itsstructure.

The syntax for defining a method in C# is as follows −

```
<Access Specifier> <Return Type> <Method Name> (Parameter List)
 {
   Method Body

  }
```

Following are the various elements of a method –

- **Access Specifier** − This determines the visibility of a variable or a method from another class.
- **Return type** − A method may return a value. The return type is the data type of thevalue the method returns. If the method is not returning any values, then the returntype is **void**.
- **Method name** − Method name is a unique identifier and it is case sensitive. It cannot be same as any other identifier declared in the class.
- **Parameter list** − Enclosed between parentheses, the parameters are used to pass and receive data from a method. The parameter list refers to the type, order, and number of the parameters of a method. Parameters are optional; that is, a methodmay contain no parameters.
- **Method body** − This contains the set of instructions needed to complete the required activity.

Program:

```
using System;

namespace CalculatorApplication
{
   class NumberManipulator
   {
      public int FindMax(int num1, int num2)
      {
         /* local variable declaration */
         int result;

         if (num1 > num2)
            result = num1;
         else
            result = num2;
         return result;
      }

      static void Main(string[] args)
      {
         /* local variable definition */
         int a = 100;
         int b = 200;
         int ret;
         NumberManipulator n = new NumberManipulator();

         //calling the FindMax method
         ret = n.FindMax(a, b);
         Console.WriteLine("Max value is : {0}", ret);
         Console.ReadLine();
      } } }
```

Output:

```
Max value is : 200
```

17.2 Passing Parameters to a Method

Sr.No.	Mechanism & Description
1	Value parameters This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
2	Reference parameters This method copies the reference to the memory location of an argument into the formal parameter. This means that changes madeto the parameter affect the argument.

17.2.1 Value parameter

- This is the default mechanism for passing parameters to a method. In this mechanism, when a method is called, a new storage location is created for eachvalue parameter.
- The values of the actual parameters are copied into them. Hence, the changes made to the parameter inside the method have no effect on the argument.

The following example demonstrates the concept –

```
using System;
namespace CalculatorApplication
{
   class NumberManipulator
   {
      public void swap(int x, int y)
      {
         int temp;

         temp = x; /* save the value of x */
         x = y;    /* put y into x */
         y = temp; /* put temp into y */
      }
      static void Main(string[] args)
      {
         NumberManipulator n = new NumberManipulator();

         /* local variable definition */
         int a = 100;
         int b = 200;

         Console.WriteLine("Before swap, value of a : {0}", a);
         Console.WriteLine("Before swap, value of b : {0}", b);

         /* calling a function to swap the values */
         n.swap(a, b);

         Console.WriteLine("After swap, value of a : {0}", a);
         Console.WriteLine("After swap, value of b : {0}", b);

         Console.ReadLine();
      }
   }
}
```

Output:

```
Before swap, value of a : 100
Before swap, value of b : 200
After swap, value of a : 100
After swap, value of b : 200
```

17.2.2 Reference parameter

- A reference parameter is a **reference to a memory location** of a variable. When you pass parameters by reference, unlike value parameters, a new storage location is not created for these parameters.
- The reference parameters represent the same memory location as the actual parameters that are supplied to the method.
- You can declare the reference parameters using the **ref** keyword.

  The following example demonstrates this –

```
using System;
namespace CalculatorApplication
{
   class NumberManipulator
   {
      public void swap(ref int x, ref int y)
      {
         int temp;

         temp = x; /* save the value of x */
         x = y;    /* put y into x */
         y = temp; /* put temp into y */
      }
      static void Main(string[] args)
      {
         NumberManipulator n = new NumberManipulator();

         /* local variable definition */
         int a = 100;
         int b = 200;

         Console.WriteLine("Before swap, value of a : {0}", a);
         Console.WriteLine("Before swap, value of b : {0}", b);

         /* calling a function to swap the values */
         n.swap(ref a, ref b);
```

```
            Console.WriteLine("After swap, value of a : {0}", a);
            Console.WriteLine("After swap, value of b : {0}", b);

            Console.ReadLine();
        } } }
```

Output:

```
Before swap, value of a : 100
Before swap, value of b : 200
After swap, value of a : 200
After swap, value of b : 100
```

18.0 Delegates

- C# delegates are similar to pointers to functions, in C or C++. A **delegate** is a reference type variable that holds the reference to a method. The reference canbe changed at runtime.
- Delegates are especially used for implementing events and the call-back methods. All delegates are implicitly derived from the **System.Delegate** class.

18.1 Declaring Delegates

- Delegate declaration determines the methods that can be referenced by the delegate.
- A delegate can refer to a method, which has the same signature as that of the delegate.

For example, consider a delegate –

public delegate int MyDelegate (string s);

- The preceding delegate can be used to reference any method that has asingle *string* parameter and returns an *int* type variable.

Syntax for delegate declaration is –

delegate <return type> <delegate-name> <parameter list>

18.2 Instantiating Delegates

- Once a delegate type is declared, a delegate object must be created withthe **new** keyword and be associated with a particular method.
- When creating a delegate, the argument passed to the **new** expression is written similar to a method call, but without the arguments to the method.

For example –

```
public delegate void printString(string s);
...
printString ps1 = new printString(WriteToScreen);
printString ps2 = new printString(WriteToFile);
```

Program

```
using System;

delegate int NumberChanger(int n);
namespace DelegateAppl
 {
 class TestDelegate
{
     static int num = 10;
 public static int AddNum(int p)
 {
        num += p;
        return num;
  }
   public static int MultNum(int q)
 {
        num *= q;
        return num;
  }

  public static int getNum()
 {
        return num;
  }
 static void Main(string[] args)
{
          //create delegate instances
       NumberChanger nc1 = new NumberChanger(AddNum);
       NumberChanger nc2 = new NumberChanger(MultNum);

       //calling the methods using the delegate objects
       nc1(25);
       Console.WriteLine("Value of Num: {0}", getNum());
       nc2(5);
       Console.WriteLine("Value of Num: {0}", getNum());
       Console.ReadKey();

 } } }
```

Output

```
Value of Num: 35
Value of Num: 175
```

18.3 Types of Delegate

i. Singlecast Delgate

ii. Multicast Delegate

18.3.1 Singlecast Delgate

In c#, a delegate that points to a single method is called a single cast delegate, and it is used to hold the reference of a single method as explained in the aboveexample.

18.3.2 Multicast Delegate

In c#, a delegate that points to multiple methods is called a multicast delegate,and it is used to hold the reference of multiple methods with a single delegate.By using "+" operator, we can add multiple method references to the delegateobject.

Program:

```
using System;

namespace Tutlane
{
    // Declare Delegate
    public delegate void SampleDelegate(int a, int b);
    class MathOperations
    {
        public void Add(int a, int b)
        {
            Console.WriteLine("Add Result: {0}", a + b);
        }
        public void Subtract(int x, int y)
        {
            Console.WriteLine("Subtract Result: {0}", x - y);
        }
        public void Multiply(int x, int y)
        {
            Console.WriteLine("Multiply Result: {0}", x * y);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("****Delegate Example****");
            MathOperations m = new MathOperations();

           // Instantiate delegate with add method
           SampleDelegate dlgt = new SampleDelegate(m.Add);

             dlgt += m.Subtract;          // dlgt= dlgt + m.subtract
            dlgt += m.Multiply;          //dlgt= dlgt + m.subtract + m.multiply

            dlgt(10, 90);

            Console.ReadKey();
        }
    }
}
```

Output:

```
****Delegate Example****
Add Result: 100
Subtract Result: -80
Multiply Result: 900
```

Questions for practice

1. Write a short note ArrayList.
2. Explain any 4 operators.
3. Explain any 5 Methods of strings.
4. Explain methods of Array and ArrayList.
5. Explain Jagged Array.
6. Write a short note on type conversion.
7. Explain foreach loop.

1.0 CLASS

- A class is simply a user-defined data type that represents both state and behavior. The state represents the properties and **behavior** is the action that objectscan perform.
- In other words, we can say that a class is the blueprint/plan/template that describes the details of an object.

1.1 Creating a class

Example:

```
class student
{
Public int roll;
Public string name;
}
```

2.0 OBJECTS

- Object is a real world entity, for example, chair, car, pen, mobile, laptop etc.
- In other words, object is an entity that has state and behavior. Here, state means data and behavior means functionality.
- Object is a runtime entity, it is created at runtime.
- Object is an instance of a class. All the members of the class can be accessed through object.

2.1 CREATING AN OBJECT

Syntax:

```
ClassName object=new ClassName();
```

Example:

```
Student s1=new student();
```

C# Object and Class Example

```
using System;
namespace  first
{
    public class Student
    {
        public int id;
        public String name;
    }
    class TestStudent
    {
        public static void Main(string[] args)
        {
            Student s1 = new Student();
            s1.id = 101;
            s1.name = "Sonoo Jaiswal";
            Console.WriteLine(s1.id);
            Console.WriteLine(s1.name);
        }
    }
}
```

NOTE:

- If Id & Name Is Not declared as public Then It Will Not Be Accessible Outside of Class.

3.0 Constructor

In C#, constructor is a special method which is invoked automatically at the time ofobject creation. It is used to initialize the data members of new object generally. The constructor in C# has the same name as class or struct.

There are 5 types of constructors in C#.

1. Default Constructor
2. Parameterized Constructor
3. Copy Constructor
4. Private Constructor
5. Static Constructor

1. Default constructor

A constructor which has no argument is known as default constructor. It is invoked at the time of creating object.

```
using System;
namespace first
{
    public class Employee
    {
        public Employee()
        {
            Console.WriteLine("Default Constructor Invoked");
        }
    }
    class TestEmployee
    {
        public static void Main(string[] args)
        {
            Employee e1 = new Employee();
            Employee e2 = new Employee();
        }
    }
}
```

Output:

```
Default Constructor Invoked
Default Constructor Invoked
```

2. Parameterized Constructor

A constructor which has parameters is called parameterized constructor. It is usedto provide different values to distinct objects

```
using System;
public class Employee
{
    public int id;
    public String name;
    public float salary;
    public Employee(int i, String n, float s)
    {
        id = i;
        name = n;
        salary = s;
    }
    public void display()
    {
        Console.WriteLine(id + " " + name + " " + salary);
    }
}
class TestEmployee
{
    public static void Main(string[] args)
    {
        Employee e1 = new Employee(101, "Sonoo", 890000f);
        Employee e2 = new Employee(102, "Mahesh", 490000f);
        e1.display();
        e2.display();
    }
}
```

3. Copy Constructor

By copying variables from another object, this constructor generates an object. Its primary purpose is to set the values of a new instance to those of an existing one.

```
using System;
namespace ConsoleApp12
{
   class student
   {
      int roll;
      string name;
      public student()
      {
         roll = 12;
```

```
            name = "neha";
        }
       public student(student a)
        {
            roll = a.roll;
            name = a.name;
        }
        public void display()
        {
            Console.Write("roll={0},name={1}", roll, name);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            student s1 = new student();
            student s2 = new student(s1);
            s2.display();
        }
    }
}
```

Here value of object s1 will be copied into s2.

4. Private Constructor in C#

A private constructor is a constructor that is created with the private specifier. Other classes cannot inherit from this class, and it is also impossible to create an instance of this class.

Note: The first important point that you need to remember is Private constructor restricts the class to be instantiated from outside the class only if it does not have any public constructor. If it has a public constructor, then we can create the instance from outside of the class. There is no restriction to creating the instance from within the same class.

Use Case: The use case of Private Constructor in C# is that if you don’t want your class to be instantiated from outside the class, then add a private constructor without any public constructor in your class.

```
using System;
namespace private_cons
{
    class student
    {
        static int roll;
        static String name;
        public student()
        {
            roll = 10;
            name = "neha";
            Console.WriteLine("default called");
            student s2=new student(6);
        }
        private student(int a)
        {
            Console.WriteLine("private called");
        }
        public void display()
        {
            Console.WriteLine("roll={0},name={1}", roll, name);
        }

    }
    class Program
    {
        static void Main(string[] args)
        {
            student s = new student();
            s.display();
        }
    }
}
```

5. Static Constructor:

This constructor is called just once in the class, and it is called when the first reference to a static member of the class is made, which is when the static constructor is called. A static constructor is used to initialize the class's static fields or data and is only used once.

```
using System;
namespace static_cons
{
    class student
    {
        static int roll;
        static String name;

        static student()
        {
            roll = 10;
            name = "neha";
            Console.Write("static called");
        }
        public void display()
        {
            Console.WriteLine("roll={0},name={1}", roll, name);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            student s1 = new student();
            student s2 = new student();
            s1.display();
            s2.display();
        }
    }
}
```

4.0 Static Data Member

- We can define class members as static using the static keyword.
- When we declare a member of a class as static, it means no matter how many objects of the class are created, there is only one copy of the staticmember.
- Static variables can be initialized outside the member function or class definition. You can also initialize static variables inside the class definition.
- Static variables are initialized by zero.

```
using System;
namespace StaticVarApplication
{
    class StaticVar
    {
        public static int num;

        public void count()
        {
            num++;
        }
     public int getNum()
        {
            return num;
        }
    }
    class StaticTester
    {
        static void Main(string[] args)
        {
            StaticVar s1 = new StaticVar();
            StaticVar s2 = new StaticVar();

            s1.count();
            s1.count();
            s1.count();

            s2.count();
            s2.count();
            s2.count();

            Console.WriteLine("Variable   num   for   s1:   {0}",   s1.getNum());
            Console.WriteLine("Variable   num   for   s2:   {0}",   s2.getNum());
            Console.ReadKey();
        }
    }
}
```

Output:

```
Variable num for s1: 6
Variable num for s2: 6
```

5.0 Static method

- You can define one or more static methods in a non-static class.
- Static methods can be called without creating an object.
- You cannot call static methods using an object.
- The static methods can only access static members.

- You cannot access non-static members of the class in the static methods.

```
using System;
class Program
{
    static int counter = 0;
    string name = "Demo Program";static
    void Display(string text)
    {
        Console.WriteLine(text);
    }
    static void Main(string[] args)
    {
        counter++; // can access static fields Display("Hello
        World!"); // can call static methods

        //name = "New Demo Program"; //Error: cannot access non-static members
}
```

6.0 STATIC CLASS

- The C# static class is like the normal class but it cannot be instantiated.
- It can have only static members.
- The advantage of static class is that it provides you guarantee that instance ofstatic class cannot be created.

6.1 Points to remember for C# static class

- C# static class contains only static members.
- C# static class cannot be instantiated.
- C# static class is sealed.

```
using System;
public static class MyMath
{
    public static float PI = 3.14f;
    //int id;  error can not create not static member
    public static int cube(int n)
  {
 return n * n * n;
 }
}
```

```
class TestMyMath
{
    public static void Main(string[] args)
    {
        Console.WriteLine("Value of PI is: " + MyMath.PI);
        Console.WriteLine("Cube of 3 is: " + MyMath.cube(3));
      // MyMath m = new MyMath(); //error, can not create object of static class
    }
}
```

6.2 Partial Class

A partial class is a special feature of C#. It provides a special ability to implement the functionality of a single class into multiple files and all these files are combined into a single class file when the application is compiled. A partial class is created by using a partial keyword. This keyword is also useful to split the functionality of methods, interfaces, or structure into multiple files.

Syntax :

```
public partial Clas_name
{
      // code
}
```

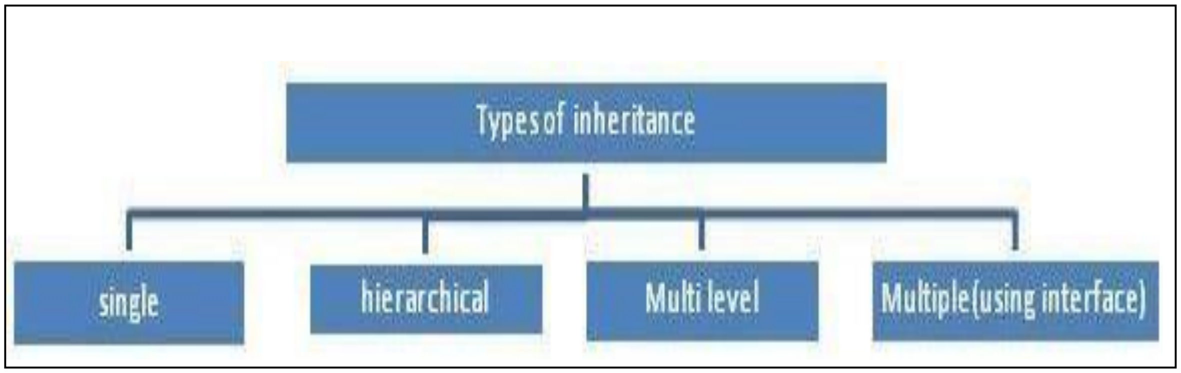
Important points:

- When you want to chop the functionality of the class, method, interface, or structure into multiple files, then you should use partial keyword and all the files are mandatory to be available at compile time for creating the final file.
- The partial modifier can only present instantly before the keywords like struct, class, and interface.
- Every part of the partial class definition should be in the same assembly and namespace, but you can use a different source file name.
- Every part of the partial class definition should have the same accessibility as private, protected, etc.
- If any part of the partial class is declared as an abstract, sealed, or base, then the whole class is declared of the same type.

7.0 INHERITANCE

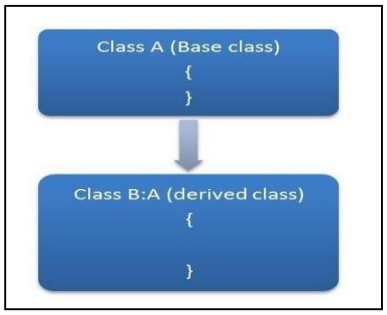
- In C#, inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically. In such way, you can reuse, extend or modify the attributes and behaviors which is defined in other class.
- In C#, the class which inherits the members of another class is called **derived class** and the class whose members are inherited is called **base** class. The derived class is the specialized class for the base class.

7.1 TYPES OF INHERITANCE



7.1.1. Single Inheritance:

It is the type of inheritance in which there is one base class and one derivedclass.

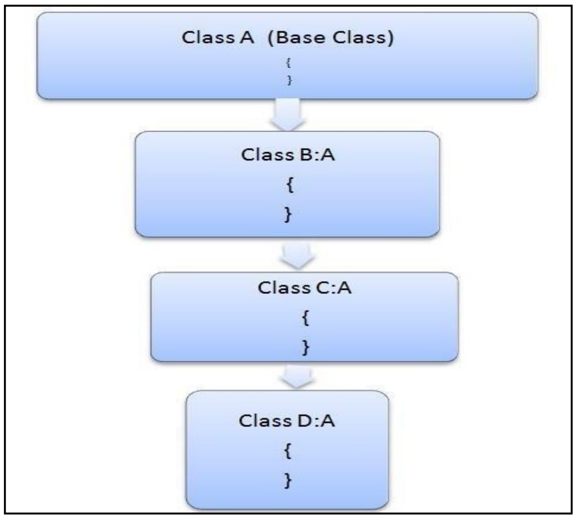


Program:

```
 using System; namespace
 Inheritance
 {
    // base class
    class Animal
    {
       public string name;
       public void display()
       {
          Console.WriteLine("I am an animal");
       }
    }
    // derived class of Animalclass
    Dog : Animal
    {
       public void getName()
       {
          Console.WriteLine("My name is " + name);
       }
}
class Program
{
    static void Main(string[] args)
    {
        // object of derived class Dog
        labrador = new Dog();

        // access field and method of base class
        labrador.name = "Rohu"; labrador.display();

        // access method from own class
        labrador.getName();
        Console.ReadKey();
        }
    } }
```

7.1.2. Hierarchical inheritance

This is the type of inheritance in which there are multiple classes derived fromone base class. This type of inheritance is used when there is a requirement ofone class feature that is needed in multiple classes.



Program:

```
using System;
namespace ConsoleApp6
{
  class number
   {
      protected int a, b;
   }
   class Addition: number
   {
      public void add(int x,int y)
      {
         a = x;
         b = y;
         Console.WriteLine("Addition=" + (a + b));
      }
   }
   class subtraction : number
   {
      public void sub(int x, int y)
      {
         a = x;
         b = y;
         Console.WriteLine("subtraction=" + (a - b));
      }
   }
   class Program
   {
      static void Main(string[] args)
      {
         int n1, n2;
         Console.WriteLine("enter 2 values:");
         n1 = Convert.ToInt32(Console.ReadLine());n2
         =       Convert.ToInt32(Console.ReadLine());
         Addition ob = new Addition();
         subtraction sb = new subtraction();
        ob.add(n1, n2);
        sb.sub(n1, n2);
    }}}
```

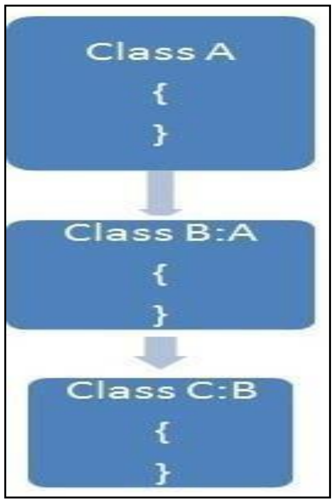
Output:

```
enter 2 values:
32
5
Addition=37
subtraction=27
```

Note: Protected members are not accessible outside of class.

7.1.3 Multilevel inheritance

When one class is derived from another, this type of inheritance is called multilevel inheritance.



Program:

```
using System; public
class Animal
{
   public void eat()
{
Console.WriteLine("Eating...");
 }
}
```

```
public class Dog : Animal
{
   public void bark()
   {
    Console.WriteLine("Barking...");
   }
}
public class BabyDog : Dog
{
 public void weep()
 {
   Console.WriteLine("Weeping...");
 }
}
class TestInheritance2
{
    public static void Main(string[] args)
    {
        BabyDog d1 = new BabyDog();
        d1.eat();
        d1.bark();
        d1.weep();
    }
}
```

8.0 Interface:

- Interface in C# is a blueprint of a class. It is like abstract class because all themethods which are declared inside the interface are abstract methods.
- It cannot have method body and cannot be instantiated.
- It is used to achieve multiple inheritance which can't be achieved by class.
- It is used to achieve fully abstraction because it cannot have method body.
- Its implementation must be provided by class.
- The class which implements the interface, must provide the implementationof all the methods declared inside the interface.

Program:

```
using System;
public interface Drawable
{
    void draw();
}
public class Rectangle : Drawable
{
    public void draw()
    {
        Console.WriteLine("drawing rectangle...");
    }
}
public class Circle : Drawable
{
    public void draw()
    {
        Console.WriteLine("drawing circle...");
    }
}
public class TestInterface
{
    public static void Main()
    {
        Drawable d;
        d = new Rectangle();
        d.draw();
        d = new Circle();
        d.draw();
    }
}
```

Note:

Interface members must be implemented with the public modifier; otherwise, the compiler will give compile-time errors.

9.0 Namespace

- Namespaces in C# are used to organize too many classes so that it can be easyto handle the application.
- In a simple C# program, we use System.Console where System is thenamespace and Console is the class.
- To access the class of a namespace, we need to use namespacename.classname.
- We can use using keyword so that we don't have to use complete name all thetime.

9.1 Creating and using a namespace

- We can create namespace using namespace keyword and also can accessthem by using the namespace or by using full qualified name.
- Full qualified name means class name must be written along with namespace name.

```
using System;
namespace number
{
    class addition
    {
        public void add()
        {
            Console.Write("Add of addition namspace\n");
        }
    }
}
namespace shape
{
    class circle
    {
        public void draw()
        {
            Console.Write("draw of shape namespace\n");
        }
       public void add()
```

```
        {
            Console.Write("Add of circle namspace\n");
        }
    }
}
namespace final
{
    class program
    {
        static void Main(String []args)
        {
            shape.circle c = new shape.circle(); number.addition
            n = new number.addition();c.draw();
            n.add();
            c.add();
        }
    }
}
```

9.2 Another way of accessing namespace

- We can also use the namespaces created by user using “using “ keyword.
- If we are using a namespace then namespace is not required along with class name.

```
using     System;
using    number;
using shape;
namespace number
{
    class addition
    {
        public void add()
        {
            Console.Write("Add of addition namspace\n");
        }
    }
}
namespace shape
{
    class circle
    {
        public void draw()
        {
            Console.Write("draw of shape namespace\n");
        }
       public void add()
        {
            Console.Write("Add of circle namspace\n");
        }
    }
}
namespace final
{
    class program
    {
        static void Main(String []args)
        {

            circle c = new circle(); addition n =
            new addition(); c.draw();
            n.add();
            c.add();
        }
    }
}
```

Note:
If namespace contains same class then ambiguity can arise, in that case we have to use namespace along with class name to avoid ambiguity;

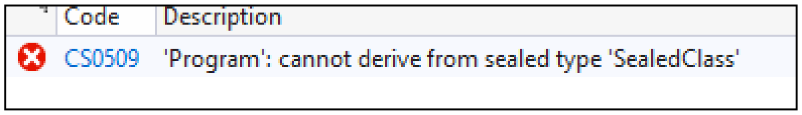
9.0 Sealed Class

- Sealed classes are used to restrict the users from inheriting the class. A classcan be sealed by using the ***sealed*** keyword.
- The keyword tells the compiler that the class is sealed, and therefore, cannotbe extended. No class can be derived from a sealed class.

Program:

```
using System;
sealed class SealedClass
{
    public int Add(int a, int b)
    {
        return a + b;
    }
}
class Program:SealedClass
{
    // Main Method
    static void Main(string[] args)
    {
        SealedClass slc = new SealedClass();int
        total = slc.Add(6, 4);
        Console.WriteLine("Total = " + total.ToString());
    }
}
```

Output



	Code	Description
❌	CS0509	'Program': cannot derive from sealed type 'SealedClass'

10.0 Polymorphism

- Polymorphism is a Greek word meaning "one name many forms." "Poly" means many, and "morph" means forms.
- In other words, one object has many forms or has one name with multiple functionalities.
- Polymorphism allows a class to have multiple implementations with the same name.

Program:

```
using System;
class Program
{
    // method does not take any parameterpublic
    void greet()
    {
        Console.WriteLine("Hello");
    }

    // method takes one string parameterpublic
    void greet(string name)
    {
        Console.WriteLine("Hello " + name);
    }

    static void Main(string[] args)
    {
        Program p1 = new Program();

        // calls method without any argument
        p1.greet();

        //calls method with an argument
        p1.greet("Tim");
    }
}
```

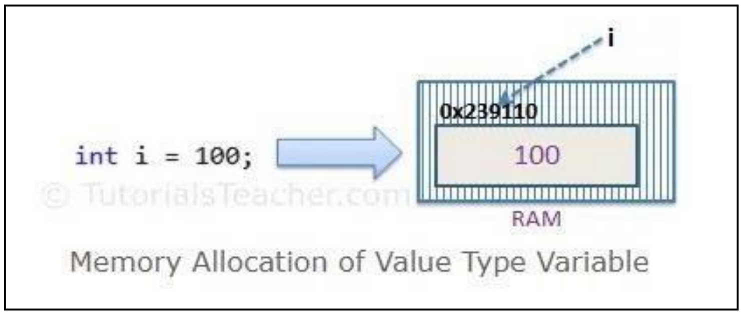
11.0 Value type and Reference type

In C#, these data types are categorized based on how they store their value inthe memory. C# includes the following categories of data types:

1. value type
2. reference type

11.1 Value Type:

- A data type is a value type if it holds a data value within its own memory space.It means the variables of these data types directly contain values.
- For example, consider integer variable int i = 100;
- The system stores 100 in the memory space allocated for the variable i. The following image illustrates how 100 is stored at some hypothetical location in the memory (0x239110) for 'i':



The following data types are all of value type:

- bool
- byte
- char
- decimal
- double
- enum
- float
- int
- long
- sbyte
- short
- struct
- uint
- ulong
- ushort

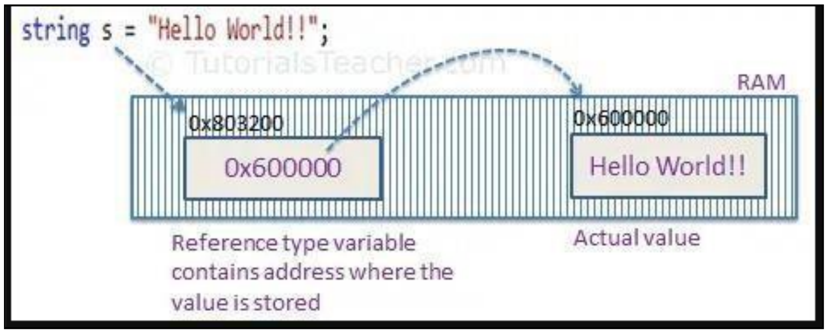
11.2 Reference Type

- Unlike value types, a reference type doesn't store its value directly.
- Instead, it stores the address where the value is being stored.
- In other words, a reference type contains a pointer to another memorylocation that holds the data.

For example, consider the following string variable:

```
string s = "Hello World!!";
```

The following image shows how the system allocates the memory for the above string variable.


string s = "Hello World!!";
© TutorialsTeacher.com
RAM
0x803200
0x600000
0x600000
Hello World!!
Reference type variable contains address where the value is stored
Actual value

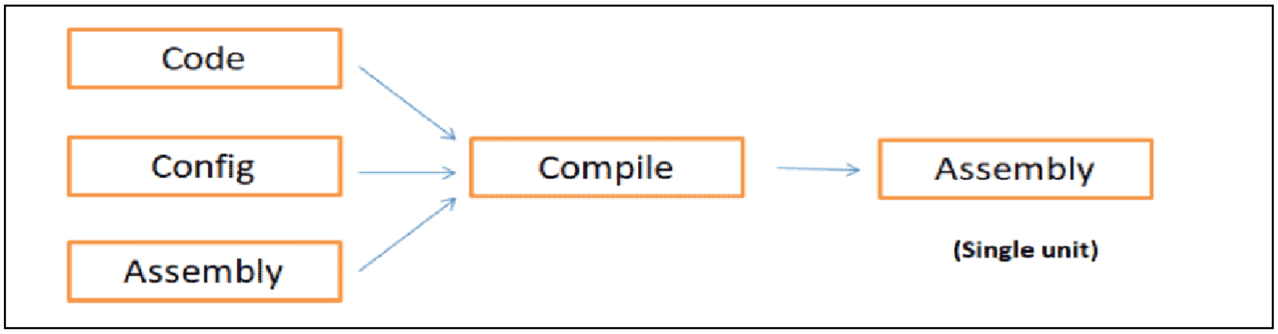
Note: String is a reference type, but it is immutable. It means once we assigned a value, it cannot be changed. If we change a string value, then the compiler creates a new string object in the memory and point a variable to the newmemory location.

The followings are reference type data types:

- String
- Arrays (even if their elements are value types)
- Class
- Delegate

12.0 Assembly

- An assembly in .NET is a single deployment unit that contains a collection of types and references. Assemblies work as a building block in a .NET application. All the resources needed by an application are contained in an assembly, such as classes, structures, interfaces, and so forth.
- An assembly is automatically created whenever a developer creates a new Windows app, web service, or class library. This assembly can be either a DLL (Dynamic Link Library) or an EXE (Executable file).
- In simple terms, an assembly in .NET is simply a precompiled chunk of code that can be run by the .NET runtime environment. A .NET program may contain one or more assemblies. Take a look at the following diagram to better grasp this concept:



- **EXE** is a short form for the word 'executable'. In any .NET application package, there will be at least one EXE file that may or may not complement one or more DLL files. Exe files have a part in the code from where the program execution starts or, put another way, has an entry point for the execution of a program. Launching an EXE file creates its own memory space by the operating system.

- **DLL** is an acronym for Dynamic Link Library, which contains functions and procedures that can be used by EXE files or libraries. As DLL is a library, you cannot execute it directly. Doing so will result in an error. Instead, a programmer can call a DLL from another program if he knows the function name and its signature in the DLL file.

12.1 What are the Types of Assemblies in .Net?

In any .NET application, assemblies can be private or public .

Private Assembly

Private assembly requires us to copy separately in all application folders where we want to use that assembly's functionalities; without copying, we cannot access the private assembly features and power. Private assembly means every time we have one, we exclusively copy into the BIN folder of each application folder.

Public Assembly

- Public assembly is not required to copy separately into all application folders. Public assembly is also called Shared Assembly. Only one copy is required in system level, there is no need to copy the assembly into the application folder.

- Public assembly should install in GAC.

- Shared assemblies (also called strong named assemblies) are copied to a single location (usually the Global assembly cache). For all calling assemblies within the same application, the same copy of the shared assembly is used from its original location. Hence, shared assemblies are not copied in the private folders of each calling assembly. Each shared assembly has a four-part name including its face name, version, public key token, and culture information. The public key token and version information makes it almost impossible for two different assemblies with the same name or for two similar assemblies with a different version to mix with each other.

13.0 ACCESS SPECIFIER/MODIFIER

- Access modifiers in C# are used to specify the scope of accessibility of a member of a class or type of the class itself.
- For example, a public class is accessible to everyone without any restrictions, while an internal class may be accessible to the assembly only.

13.1 C# provides five types of access specifiers.

1. Public
2. Protected
3. Internal
4. Protected internal
5. Private

Modifier	Description
public	There are no restrictions on accessing public members.
private	Access is limited to within the class definition. This is the default access modifier type if none is formally specified
protected	Access is limited to within the class definition and any class that inherits fromthe class
internal	Access is limited exclusively to classes defined within the current project assembly
protected internal	Access is limited to the current assembly and types derived from the containingclass. All members in current project and all members in derived class can access the variables.

Boxing and Unboxing:

- **Boxing:** Boxing is the process of converting a value type (like int, double, struct) to a reference type (object). When a value type is boxed, a new object is allocated to the heap, and the value is copied into it.

- **Unboxing:** Unboxing is the reverse process of boxing, where a value is extracted from an object. It involves explicitly converting a reference type (object) into a value type. This operation also involves a copy operation, where the value is copied from the heap into the stack.

Boxing	Unboxing
It convert value type into an object type.	It convert an object type into value type.
Boxing is an implicit conversion process.	Unboxing is the explicit conversion process.
Here, the value stored on the stack copied to the object stored on the heap memory.	Here, the object stored on the heap memory copied to the value stored on the stack .

Example:

```
// C# program to illustrate Boxing
using System;

public class GFG {
static public void Main()
{
int val = 2019;

// Boxing
object o = val;

// Change the value of val
val = 2000;

Console.WriteLine("Value type of val is {0}", val);
Console.WriteLine("Object type of val is {0}", o)
}
}
```

Output:
```
Value type of val is 2000
Object type of val is 2019
```

```
// C# program to illustrate Unboxing
using System;

public class GFG {
    static public void Main()
    {
        int val = 2019;

        // Boxing
        object o = val;

        // Unboxing
        int x = (int)o;

        Console.WriteLine("Value of o is {0}", o);
        Console.WriteLine("Value of x is {0}", x);
    }
}
```

Output:
```
Value of o is 2019
```