UNIT - 2

Variable in C#

- Variable is a named memory location, whose value can change during execution of a program.
- We know that while developing any software application using c#; we will be dealing with different types of data and data values. In order to store and process different types of data & data values computer uses its memory (RAM). To allocate a chunk of memory and access it within a C# program, we need to declare a variable.
- **Declaring a variable:**

-means allocating a memory location for some data.

- **Initializing a variable:**

- means assigning the initial value to that allocated memory location.

datatype nameOfVariable; //declaration of a single variable

Ex:

Int a;

Int b;

Note: In C# default value of variable depends upon the type of a variable.

Default value of

-numeric type is 0

-bool data type is false

-char data type is '\0' character

-reference type is null

- Before using local variable, it has to be assigned or initialized with some meaningful value.
- **Syntax for assigning value to a variable:**

Ex:

a = 0;

b = 10;

datatype nameOfVariable = ivalue; // declaration & initialization of a single variable.

int a = 0;

int b = 10;

- **Syntax for declaring multiple variable:**

datatype nameOfVariable1, nameOfVariable2,……;// declaration of multiple variables

Ex: int a, b;

- **Syntax for declaring & initializing multiple variable:**

datatype nameOfVariable1 = ivalue, nameOfVariable2 = ivalue,……;// declaration & initialization of multiple variables

Ex: int a = 0, b = 10;

Rules to Declare a C# Variables:

We can define a variable name with the combination of alphabets, numbers and underscore.

- A variable name must always starts with either alphabets or underscore but not with numbers.
- While defining the variable, no white space is allowed within the variable name.
- We should not use any reserve keywords such as int, float, char, etc. for variable name.
- In c#, once the variable is declared with a particular data type, it cannot be re-declared with a new type and we shouldn't assign a value that is not compatible with the declared type.

Example

```
using System;
namespace VariableDemo
{
   class Program
   {
      static void Main(string[] args)
      {
         int a=0;
         int b=10;
         Console.WriteLine(a);
         Console.WriteLine(b);
         Console.ReadLine();
      }
   }
}
```

C# Data Types

Understanding Value Types and Reference Types.

- In C# programming language, Data Types are used to define a type of data the variable can hold such as integer, float, string, etc. in our application.
- it's mandatory to define a variable with required data type to indicate what type of data that variable can hold in our application

Value Type

- All fixed length data types int, float, char etc. will comes under the category of value type.

Reference Type

- All variable length data types like string and object will comes under the category of reference types.

Syntax of Defining C# Data Types

[Data Type] [Variable Name];

[Data Type] [Variable Name] = [Value];

Eg: int a;

int a=10;

[Data Type] - It's a type of data the variable can hold such as integer, string, decimal, etc.

[Variable Name] - It's a name of the variable to hold the values in our application.

[Value] - Assigning a required value to the variable.

Example:

```
using System;
namespace DatatypeDemo
{
        class  Program
        {
                static  void  Main(string[]  args)
                {
                        byte b = 1;
                        int n = 10;
                        float f = 1.2f;
                        double h = 3.123;
                        decimal d = 3.142m;
                        char c = 'A';
                        bool l = false;
                        Console.WriteLine(b);
                        Console.WriteLine(n);
                        Console.WriteLine(f);
                        Console.WriteLine(h);
                        Console.WriteLine(d);
                        Console.WriteLine(c);
                        Console.WriteLine(l);
                        Console.ReadLine();
```

```
            }
        }
    }
```

C# Operators (Arithmetic, Relational, Logical, Assignment)

C# has rich set of built-in operators and provides the following type of operators:

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators

Arithmetic Operators

Operator	Name	Description	Example (a = 6, b = 3)
+	Addition	It adds two operands.	a + b = 9
-	Subtraction	It subtract two operands.	a - b = 3
*	Multiplication	It multiplies two operands.	a * b = 18
/	Division	It divides numerator by de-numerator.	a / b = 2
%	Modulo	It returns a remainder as result.	a % b = 0

```
using System;
namespace OURcreativeINFO
{
    class Program
    {
        static void Main(string[] args)
            {
                int result;
                int x = 20, y = 10;
                result = (x + y);
```

```
                Console.WriteLine("Addition Operator: " + result);
                result = (x - y);
                Console.WriteLine("Subtraction Operator: " + result);
                result = (x * y);
                Console.WriteLine("Multiplication Operator: "+ result);
                result = (x / y);
                Console.WriteLine("Division Operator: " + result);
                 result = (x % y);
                Console.WriteLine("Modulo Operator: " + result);
                Console.ReadLine();
            }
        }
    }
```

Relational Operators

Relational operators are used for comparison purpose in conditional **.**

Example:

```
using System;
namespace OURcreativeINFO
{
    class  Program
    {
        static void Main(string[]  args)
        {
            bool result;
            int x = 10, y = 20;
            result = (x == y);
            Console.WriteLine("Equal to Operator: " + result);
            result = (x > y);
            Console.WriteLine("Greater than Operator: " + result);
            result = (x <= y);
            Console.WriteLine("Lesser than or Equal to: "+ result);
            result = (x != y);
            Console.WriteLine("Not Equal to Operator: " + result);
            Console.WriteLine("Press Enter Key to Exit..");
            }
        }
    }
```

Logical and Bitwise Operator

- If we use Logical AND, OR operators in c# applications, those will return the result like as shown below for different inputs.

Operand1	Operand2	AND	OR
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

Example:

- Some examples of the usage of logical expression are:

int age=56;

Salary=9000;

- If (age>55 && salary<1000)
- If (number <0|| number >100)

Conditional Operators (Ternary Operator)

- Ternary operator is a Conditional operator in C#. It takes three arguments and evaluates a Boolean expression.
- It is the replacement of if else condition.
- For example –

```
int a=10, b=7;
var result = a > b ? “a is greater than b”:”a is less than b”;
```

- For example –

```
b = (a == 1) ? 20 : 30;
```

- Above, if the first operand evaluates to true (1), the second operand is evaluated. If the first operand evaluates to false (0), the third operand is evaluated

Example:

```
using System;
namespace DEMO
```

```
{
        class Program
        {
                static void Main(string[] args)
                {
                        int a, b;
                        a = 10;
                        b = (a == 1) ? 20 : 30;
                        Console.WriteLine("Value of b is {0}", b);
                        b = (a == 10) ? 20 : 30;
                        Console.WriteLine("Value of b is {0}", b);
                }
        }
}
```

Assignment Operators

Assignment operators are use to assign values to variables.

For example, we can declare and assign a value to the variable using assignment operator (=) like as shown below.

int a=10;

INCREMENT AND DECREMENT OPERATORS:

- **PREFIX NOTAION:** The increment operator ++ if used as prefix on a variable, the value of variable gets incremented by 1. After that the value is returned unlike Postfix operator. It is called Prefix increment operator. In the same way the prefix decrement operator works but it decrements by 1.

```
using System;
namespace OCI
{
        class Program
        {
                static void Main(string[] args)
                {
                        int a, b;
                        a = 50;
                        Console.WriteLine(++a);
                        b = a;
                        Console.WriteLine(a);
                        Console.WriteLine(b);
                }
        }
```

Postfix Operator:

- The increment operator ++ if used as postfix on a variable, the value of variable is first returned and then gets incremented by 1. It is called Postfix increment operator. In the same way the decrement operator works but it decrements by 1.

```
namespace KLEBCA
{
        class Program
        {
                static void Main(string[] args)
                {
                        int a, b;
                        a = 10;
                        Console.WriteLine(a++);
                        b = a;
                         Console.WriteLine(a);
                        Console.WriteLine(b);
                }
        }
}
```

Flow control structures in C#

- if statement
- if else
- else if
- While
- do while
- Switch
- For
- For each

C# if statement

- Use the if statement to specify a block of C# code to be executed if a condition is True.
- The if statement has the following general form:

```
if (expression)
{
    statement;
}
```

Example:

```
class Program
```

```
{
   static void Main(string[] args)
   {
      int x,y;
      x= 10;
      y=15;
      if(x<y)
      {
            Console.WriteLine("x is less than y");
       }
     }
  }
```

if else statement

Use the else statement to specify a block of code to be executed if the condition is False.

Syntax:

if (*condition*)

{ *// block of code to be executed if the condition is True*

}

else

{ *// block of code to be executed if the condition is False*

}

Example:

```
class Program
{
    static void Main(string[] args)
    {
       int time = 20;
       if(time<18)
       {
           Console.WriteLine("Good evening");
       }
       else
       {
           Console.WriteLine("Good night");
           Console.ReadLine();
        }
    }
```

The else if Statement

Example:

```
class Program
  {
    static void Main(string[] args)
    {
       int time = 22;
       if (time < 10)
         {
                  Console.WriteLine("Good morning.");
         }
         else if (time < 20)
         {
                  Console.WriteLine("Good day.");
         }
         else
         {
                  Console.WriteLine("Good evening.");
          }
      Console.ReadLine();
  }
 }
```

C# While Loop

The while loop loops through a block of code as long as a specified condition is True:

```
while (condition)
{
// code block to be executed }
```

Example:

```
using system;
class Program
  {
    static void Main(string[] args)
    {
   int i = 0;
   while (i < 5)
   {
    Console.WriteLine(i);
    i++;
   }
  }
 }
```

The Do/While Loop

The do/while loop is a variant of the while loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true

```
do {
// code block to be executed
}
while (condition);
```

Example:

```
using system;
class Program
  {
    static void Main(string[] args)
    {
       int i = 0;
        do
        {
         Console.WriteLine(i);
          i++;
        }
         while (i < 5);
         }
    }
  }
```

C# Switch Statements

Use the switch statement to select one of many code blocks to be executed.

```
switch(expression)
      {
      case x:
          // code block
          break;
      case y:
          // code block
          break;
      default:
```

```
        // code block
    break;
}
```

Switch case

- The switch expression is evaluated once
- The value of the expression is compared with the values of each case
- If there is a match, the associated block of code is executed.

```
using system;
class Program
  {
    static void Main(string[] args)
    {
      int day = 4;
      switch (day)
      {
                case 1:
                Console.WriteLine("Monday");
                break;
                case 2:
                Console.WriteLine("Tuesday");
                break;
                case 3:
                Console.WriteLine("Wednesday");
                break;
                case 4:
                Console.WriteLine("Thursday");
                break;
                case 5:
                Console.WriteLine("Friday");
                break;
                case 6:
                Console.WriteLine("Saturday");
                break;
                case 7:
                Console.WriteLine("Sunday");
                break;
    Console.ReadLine();
          }
          }
  }
}
```

For loop

- When you know exactly how many times you want to loop through a block of code, use the for loop instead of a while loop:

Syntax

for (*statement 1*; *statement 2*; *statement 3*)

```
{
    // code block to be executed
}
```

Example:

```
using system;
Class Program
{
      static void Main(string[] args)
      {
            for (int i = 0; i < 5; i++)
            {
                  Console.WriteLine(i);
             }
       }
}
```

for each loop

- There is also a for each loop, which is used exclusively to loop through elements in an array:

```
foreach (type variableName in arrayName)
{
      // code block to be executed
}
```

- The foreach loop in C# uses the 'in' keyword to iterate over the iterable item.
- The **in** keyword selects an item from the collection for the iteration and stores it in a variable called the loop variable, and the value of the loop variable changes in every iteration.
- For example, the first iteration process will have the first item of the collection stored; the second iteration will have the second item of the collection stored, and so on.

The number of iterations of the C# foreach loop equals the total items in the collection.

Example:

```
using system;
class Program
  {
    static void Main(string[] args)
    {
       string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
     foreach (string i in cars)
       {
                  Console.WriteLine(i);
```

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

Object and Classes

Since C# is an object-oriented language, the program is designed using objects and classes in C#.

Concept of a class:

In C#, a class is a group of similar objects. It is a template from which objects are created. It can have fields, methods, constructors, etc.

Let's see an example of a C# class that has two fields only.

```
class Student
{
   int id; //field or data member
   String name; //field or data member
}
```

Object

In C#, an Object is a real-world entity, for example, chair, car, pen, mobile, laptop etc.

In other words, the object is an entity that has a state and behaviour. Here, state means data and behaviour 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.

Let's see an example:1 to create object using new keyword.

```
Student s1 = new Student();//creating an object of Student
```

In this example, Student is the type and s1 is the reference variable that refers to the instance of the Student class. The new keyword allocates memory at runtime.

C# Object and Class Example

Let's see an example of a class that has two fields: id and name. It creates an instance of the class, initializes the object, and prints the object value.

C# Class Example 1: Having Main() in same class

```
using System;
  public class Student
   {
      int id; //data member (also instance variable)
      String name; //data member (also instance variable)
      public static void Main(string[] args)
      {
         Student s1 = new Student();//creating an object of Student
         s1.id = 101;
         s1.name = "Virat";
         Console.WriteLine(s1.id);
         Console.WriteLine(s1.name);
      }
   }
```

C# Class Example 2: Having Main() in another class

Let's see another example of a class where we are having Main() method in another class. In such cases, a class must be public.

```
using System;
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 = "Virat";
                Console.WriteLine(s1.id);
                Console.WriteLine(s1.name);
        }
}
```

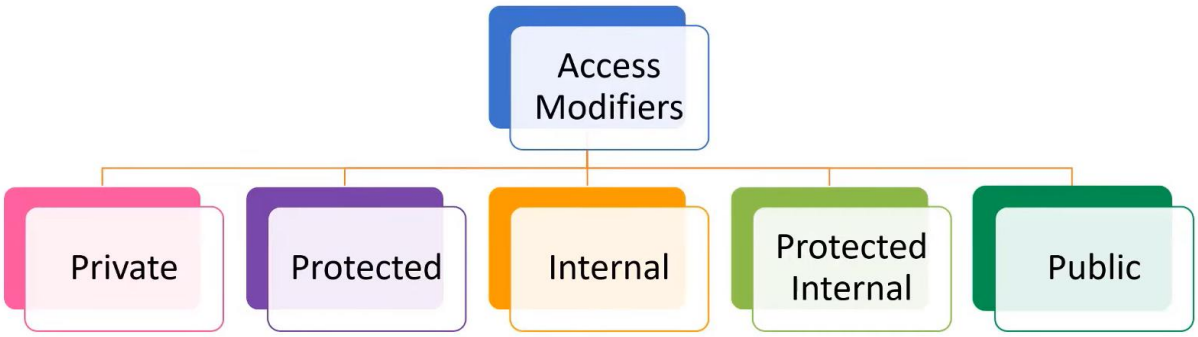
Fields

Fields are the data contained in the class. Fields may be implicit data types, objects of some other class, enumerations, structs or delegates. In the example below, we define a class named Student containing a student's name, age, marks in maths, marks in English, marks in science, total marks, obtained marks and a percentage.

```
class Student
{
        // fields contained in Student class
        string name;
        int age;
        int marksInMaths;
        int marksInEnglish;
        int marksInScience;
        int totalMarks = 300; // initialization
        int obtainedMarks;
        double percentage;
}
```

You can also initialize the fields with the initial values as we did in totalMarks in the example above. If you don't initialize the members of the class, they will be initialized with their default values.

ACCESS MODIFIERS:



Access Modifiers are the keywords which are used to define an accessibility level for all types and type members.

By specifying an access level for all types and type members, we can control that whether they can be accessed in other classes or in current assembly or in other assemblies based on our requirements.

Following are the different type of access modifiers available in c# programming language.

- Public
- Private
- Protected
- Internal
- Protected internal

Access Modifier	Description
public	It is used to specifies that access is not restricted.
private	It is used to specifies that access is limited to the containing type.
protected	It is used to specifies that access is limited to the containing type or types derived from the containing class.
internal	It is used to specifies that access is limited to current assembly.
protected internal	It is used to specifies that access is limited to the current assembly or types derived from the containing class.

Static members of the class:

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 static member.

The keyword static implies that only one instance of the member exists for a class. Static variables are used for defining constants because their values can be retrieved by invoking the class without creating an instance of it. Static variables can be initialized outside the member function or class definition. You can also initialize static variables inside the class definition.

The following is an example –

Example

```
using System;
namespace staticmembers
 {
   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:" +s1.getNum());
      Console.WriteLine("Variable num for s2:" +s2.getNum());
      Console.ReadKey();
    }
  }
}
```

Output:

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

C# Constructor

Constructor is a special method present under a class responsible for initializing the variables of the class.

The name of the constructor method is exactly the same name of the class in which it was present.

Each and every class must have the constructor if we want to create the instance of that class.

There can be two types of constructors in C#.

- Default constructor
- Parameterized constructor

C# Default Constructor

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

C# Default Constructor Example: Having Main() within class

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

Output:

```
Default Constructor Invoked

Default Constructor Invoked
```

C# Default Constructor Example: Having Main() in another class

Let's see another example of a default constructor where we are having Main() method in another class.

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

Output:

Default Constructor Invoked

Default Constructor Invoked

C# Parameterized Constructor

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

```
using System;
public class Employee
{
   public int id;
   public String name;
   public Employee(int i, String n)
   {
      id = i;
      name = n;
   }
   public void display()
   {
      Console.WriteLine(id + " " + name);
   }
}

class TestEmployee
{
   public static void Main(string[] args)
   {
      Employee e1 = new Employee(101, "ABC");
      Employee e2 = new Employee(102, "EFG");
      e1.display();
```

```
        e2.display();
        Console.ReadLine();
    }
}
```

Output:

101 ABC
102 EFG

Destructors

A destructor works opposite to constructor, It destructs the objects of classes. It can be defined only once in a class. Like constructors, it is invoked automatically.

Note: C# destructor cannot have parameters. Moreover, modifiers can't be applied on destructors.

Important Points:

- A Destructor is unique to its class i.e. there cannot be more than one destructor in a class.
- A Destructor has no return type and has exactly the same name as the class name (Including the same case).
- It is distinguished apart from a constructor because of the *Tilde symbol (~)* prefixed to its name.
- A Destructor does not accept any parameters and modifiers.
- It cannot be defined in Structures. It is only used with classes.
- It cannot be overloaded or inherited.
- It is called when the program exits.
- Internally, Destructor called the Finalize method on the base class of object.

Let's see an example of constructor and destructor in C# which is called automatically.

```
using System;
  public class Employee
  {
     public Employee()
     {
        Console.WriteLine("Constructor Invoked");
     }
     ~Employee()
```

```
    {
        Console.WriteLine("Destructor Invoked");
    }
}
class TestEmployee{
    public static void Main(string[] args)
    {
        Employee e1 = new Employee();
        Employee e2 = new Employee();
    }
}
```

Output:

```
Constructor Invoked
Constructor Invoked
Destructor Invoked
Destructor Invoked
```

Note: Destructor can't be public. We can't apply any modifier on destructors.

Method Overloading

Having two or more methods with same name but different in parameters, is known as method overloading in C#.

The **advantage** of method overloading is that it increases the readability of the program because you don't need to use different names for same action.

You can perform method overloading in C# by two ways:

1. By changing number of arguments
2. By changing data type of the arguments

C# Method Overloading Example: By changing no. of arguments

Let's see the simple example of method overloading where we are changing number of arguments of add() method.

```
using System;
public class Cal
{
```

```
  public static int add(int a,int b)
  {
    return a + b;
  }
  public static int add(int a, int b, int c)
  {
    return a + b + c;
  }
}
public class TestMemberOverloading
{
  public static void Main()
  {
    Console.WriteLine(Cal.add(12, 23));
    Console.WriteLine(Cal.add(12, 23, 25));
  }
}
```

Output:

```
35
65
```

C# Member Overloading Example: By changing data type of arguments

Let's see the another example of method overloading where we are changing data type of arguments.

```
using System;
public class Cal
{
  public static int add(int a, int b)
  {
    return a + b;
  }
  public static float add(float a, float b)
  {
    return a + b;
  }
```

```
}
public class TestMemberOverloading
{
   public static void Main()
   {
      Console.WriteLine(Cal.add(12, 23));
      Console.WriteLine(Cal.add(12.4f,21.3f));
   }
}
```

Output:

35
33.7

Thank You

(Our Creative Info)