C# Introduction:

It is an object-oriented programming language created by Microsoft that runs on the .NET Framework.
it is a high-level language

C# is used for:

- Mobile applications
- Desktop applications
- Web applications
- Web services
- Web sites
- Games
- VR
- Database applications

Create first C# program 'Hello world':

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

Main – It is the entry point of the execute the program
Console.WriteLine() – Write a line of text out onto the screen.
Console.ReadLine() – Pauses the screen and write something on the screen

Datatypes and Variables:

- Variables is a container to store the data value.
- Use the variable name in the program
- string used to store sequence of the character
- int used to store the numeric value
- char used to store the character value
- float, double and decimal used to store a fraction value
    - accurate --> decimal > double > float
- bool stored only true or false

Operators:

- It allows us to perform different kinds of operations on operands.

- **Arithmetic Operators**

Operator	Name	Description	Example
+	Addition	Adds together two values	x + y
-	Subtraction	Subtracts one value from another	x - y
*	Multiplication	Multiplies two values	x * y
/	Division	Divides one value by another	x / y
%	Modulus	Returns the division remainder	x % y
++	Increment	Increases the value of a variable by 1	x++
--	Decrement	Decreases the value of a variable by 1	x--

- **Conditional Operators**

Operator	Name	Example
==	Equal to	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

- **Logical Operators**

Operator	Name	Description	Example
&&	Logical and	Returns True if both statements are true	x < 5 && x < 10
||	Logical or	Returns True if one of the statements is true	x < 5 || x < 4
!	Logical not	Reverse the result, returns False if the result is true	!(x < 5 && x < 10)

- **Bitwise Operators**

Operator	Description	Example
&	bitwise and	a & b
|	bitwise or	a | b
^	bitwise xor	a ^ b
<<	left shift	a << 1
>>	right shift	a >> 1

- **Assignments Operators**

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Statements:

- The actions that a program takes are expressed.
- A programming language uses control statements to control the flow of execution of program based on certain conditions.

if statement

```
If(condition){
      // code to be executed
}
```

If-else statement

```
If(condition){
      // code if condition is true
}
else{
      // code if condition is false
}
```

If-else-if ladder statement

```
If(condition1){
      // code if condition1 is true
}
else if (condition2){
      // code if condition2 is true
}
else{
      // code if all condition is false
}
```

switch statement

```
switch(expression){
    case value1:   // statement sequence
                   break;
    case value2:   // statement sequence
                   break;
    default:   //default statement sequence
}
```

while loop

```
 int i = 1;
 while(i<=4){
     Console.WriteLine("The value of i is "+i++);
 }
```

for loop

```
for(int j=0;j<=5;j++){
    if(j==3)
        continue;
    Console.WriteLine("The value of j is "+j);
}
```

do while

```
i=1;
do{
   Console.WriteLine("The value of i is "+i++);
}while(i<=4);
```

Array:

- An array is a group of like-typed variables that are referred to by a common name.
- **Length** of the array specifies the number of elements present in the array.
- all arrays are dynamically allocated.
- C# array is an object of base type **System.Array**.
- Only Declaration of an array doesn't allocate memory to the array. For that array must be initialized.
- int[] luckyNumbers = [1,34,6,3,322,232];
- string[] friends = new string[3];
- int[] intArray = new int[5]{1, 2, 3, 4, 5};
- int[,] intTwoDArray = new int[,]{{1,2},{3,4}};
- int[,] intarray_d = new int[4, 2] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
- An array whose elements are arrays is known as **Jagged arrays;** it means "**array of arrays**".
- int[][] arr = new int[2][];
- arr[0] = new int[2]{2,3};
- arr[1] = new int[3]{32,5,65};
- int[][] arrAnother = {
    new int[]{1,2,3,4,5},
    new int[]{6,7,8}
  };

Clear()	Sets a range of elements in an array to the default value of each element type.
Clone()	Creates a shallow copy of the Array.
Copy()	Copies a range of elements in one Array to another Array and performs type casting and boxing as required.

CopyTo()	Copies all the elements of the current one-dimensional array to the specified one-dimensional array.
Equals()	Determines whether the specified object is equal to the current object.
ForEach()	Performs the specified action on each element of the specified array.
GetType()	Gets the Type of the current instance.
GetValue()	Gets the value of the specified element in the current Array.
IndexOf()	Searches for the specified object and returns the index of its first occurrence in a one-dimensional array or in a range of elements in the array.
Resize()	Changes the number of elements of a one-dimensional array to the specified new size, use keyword **ref.**
Reverse()	Reverses the order of the elements in a one-dimensional Array or in a portion of the Array.
Sort()	Sorts the elements in a one-dimensional array.
ToString()	Returns a string that represents the current object.

Methods:

- Methods are generally the block of codes or statements in a program that gives the user the ability to **reuse** the same code.
- it provides a better **readability** of code.

OOP:

- OOP stands for Object-Oriented Programming.
- OOP is faster and easier to execute.
- OOP helps to keep the C# code DRY “Don’t Repeat Yourself”, and makes the code easier to maintain, modify and debug.
- class is a template for objects, and an object is an instance of a class.

Classes and Objects:

- A Class is like an object constructor, or a "blueprint" for creating objects.

```
class Customer
{
    string _firstName, _lastName;
}
Customer customer = new Customer();
```

- Fields and methods inside classes are often referred to as "Class Members":

Constructors:

- A constructor is a special method that is used to initialize objects.
- it is called when an object of a class is created.
- It can be used to set initial values for fields.

```
class Customer
{
  string _firstName, _lastName;            // private member

  public Customer()
  {
    Console.WriteLine("No name is assigned");
  }
  public Customer(string firstName, string lastName)
  {
    this._firstName = firstName;
    this._lastName = lastName;
  }

}

public static void Main(string[] args)
{
  Customer c1 = new Customer("Dev", "Nakum");
  Customer c2 = new Customer();
}
```

Access Modifiers:

- To control the visibility of class members (the security level of each individual class and class member).
- To achieve "**Encapsulation**" - which is the process of making sure that "sensitive" data is hidden from users. This is done by declaring fields as **private**.

Modifier	Description
`public`	The code is accessible for all classes
`private`	The code is only accessible within the same class
`protected`	The code is accessible within the same class, or in a class that is inherited from that class. You will learn more about inheritance in a later chapter
`internal`	The code is only accessible within its own assembly, but not from another assembly. You will learn more about this in a later chapter

Encapsulation:

- The meaning of Encapsulation is to make sure that "**sensitive**" data is hidden from users.
- declare fields/variables as private.
- provide public get and set methods, through properties, to access and update the value of a private field.

```
public string FirstName
{
   get { return _firstName; }
   set { this._firstName = value; }
}
```

or

```
public string FirstName { get; set;}
```

- Fields can be made **read-only** (if you only use the get method), or **write-only** (if you only use the set method)
- Increased security of data

Inheritance:

- Derived and Base Class
- **Derived Class (child) -** the class that inherits from another class
- **Base Class (parent)** - the class being inherited from
- It is useful for code reusability: reuse fields and methods of an existing class when you create a new class.
- If you don't want other classes to inherit from a class, use the **sealed** keyword

```
class Customer
{
   string _firstName, _lastName;

   public Customer()
   {
      Console.WriteLine("No name is assigned");
   }
```

```
    public Customer(string firstName, string lastName)
    {
        this._firstName = firstName;
        this._lastName = lastName;
    }

    public string FirstName
    {
        get { return _firstName; }
        set { this._firstName = value; }
    }

    public void PrintFullName()
    {
        Console.WriteLine("Full name is " + this._firstName + " " + this._lastName);
    }

}

class VIPCoustomer : Customer
{
    public bool isVIP;

    public VIPCoustomer(string firstName, string lastName) : base(firstName, lastName)
    {
        this.isVIP = true;
    }

    public void PrintFullName()
    {
        base.PrintFullName();
        Console.WriteLine("Thank you for using our VIP service !!");
    }

}
```

Polymorphism:

- Polymorphism means "**many forms**", and it occurs when we have many classes that are related to each other by inheritance.

```
class Animal
{
    public virtual void AnimalSound()
    {
        Console.WriteLine("The animal makes sound");
    }
}
```

```
class Pig : Animal
{
  public override void AnimalSound()
  {
    Console.WriteLine("The pig says: wee wee");
  }
}

class Dog : Animal
{
  public override void AnimalSound()
  {
    Console.WriteLine("The dog says: bow bow");
  }
}
```

Abstraction:

- Data abstraction is the process of hiding certain details and showing only essential information to the user.
- Abstraction can be achieved with either abstract classes or interfaces
- The abstract keyword is used for classes and methods:
  - **Abstract class**: is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
  - **Abstract method:** can only be used in an abstract class, and it does not have a body. The body is provided by the derived class (inherited from).
- An abstract class can have both abstract and regular.

```
abstract class Shape
{
  public abstract void Draw();
}
```

- To access the abstract class, it must be inherited from another class.

```
class Circle : Shape
{
  public override void Draw()
  {
    Console.WriteLine("Circle is draw");
  }
}
```

Interface:

- Another way to achieve abstraction in C#, is with interfaces.
- An interface is a **completely** "abstract class", which can only contain abstract methods and properties (with empty bodies)

```
interface IShape
{
  void Draw();
```

```
        int Area { get; }
    }
```

- By default, members of an interface are abstract and public.
- do not have to use the override keyword when implementing an interface

```
class Square : IShape
{
  int _length;
  public Square(int length)
  {
    this._length = length;
  }

  public void Draw()
  {
    Console.WriteLine("Drawing a Square");
  }

  public int Area
  {
    get { return _length * _length; }
  }
}
```

- Like abstract classes, interfaces cannot be used to create objects.
- Interface methods do not have a body - the body is provided by the "implement" class.
- On implementation of an interface, you must override all of its methods.
- Interface members are by default abstract and public.
- An interface cannot contain a constructor.

Collections:

- It contains a set of classes to contain elements in a generalized manner.
- With the help of collections, the user can perform several operations on objects like the store, update, delete, retrieve, search, sort etc.

- **ArrayList** □ It is a dynamic array means the size of the array is not fixed, it can increase and decrease at runtime.
- **Hashtable** □ It represents a collection of key-and-value pairs that are organized based on the hash code of the key.
- **Queue** □ It represents a first-in, first out collection of objects. It is used when you need a first-in, first-out access to items.
- **Stack** □ It is a linear data structure. It follows the LIFO(Last In, First Out) pattern for Input/output.

- **Dictionary<TKey,TValue>** □ It stores key/value pairs and provides functionality similar to that found in the non-generic Hashtable class.

- **List<T>** □ It is a dynamic array that provides functionality similar to that found in the non-generic ArrayList class.
- **Queue<T>** □ A first-in, first-out list and provides functionality similar to that found in the non-generic Queue class.
- **SortedList<TKey,TValue>** □ It is a sorted list of key/value pairs and provides functionality similar to that found in the non-generic SortedList class.
- **Stack<T>** □ It is a first-in, last-out list and provides functionality similar to that found in the non-generic Stack class.
- **HashSet<T>** □ It is an unordered collection of the unique elements. It prevents duplicates from being inserted in the collection.

Enum:

- An enum is a special "class" that represents a group of constants.
- To create an enum, use the enum keyword (instead of class or interface), and separate the enum items with a comma
- By default, the first item of an enum has the value 0. The second has the value 1, and so on.
- enum has fixed set of constants
- enum improves type safety
- enum can be traversed

```
enum enmMonth
  {
    January,
    February,
    March,
    April,
    May,
    June,
    July,
    August,
    September,
    October,
    November,
    December
  }
```

access enum items with the **dot** syntax → Month = (int)enmMonth.March

```
// get the month names from enum and store it
 string[] MonthsName = Enum.GetNames(typeof(enmMonth));
```

Data Table:

- The C# DataTable is defined as the class which contains a number of rows and columns for storing and retrieving the data from both the memory and the database.

```
// create the instance of dataTable
DataTable dataTable = new DataTable();

// create the instance of dataColumn
DataColumn columnId = new DataColumn("Id", typeof(int));
DataColumn columnName = new DataColumn("Name", typeof(string));

// Add column instance to Data Table
dataTable.Columns.Add(columnId);
dataTable.Columns.Add(columnName);

// add primary key
DataColumn[] primaryKey = { dataTable.Columns["Id"] };
dataTable.PrimaryKey = primaryKey;

// create the instance of dataRow and assign the data into row
DataRow row = dataTable.NewRow();
row["Id"] = 1;
row["Name"] = "Dev";
dataTable.Rows.Add(row);


// Iterating the data Table
 foreach (DataRow dataRow in dataTable.Rows)
 {
    Console.WriteLine($"Id : {dataRow["Id"]}");
    Console.WriteLine($"Name : {dataRow["Name"]}");
  }
```

```
// modify the data

 DataRow modifyRow = dataTable.Rows.Find(1);

 if( modifyRow != null )
 {
    modifyRow["Name"] = “Kishan";
 }

  DataRow deleteRow = dataTable.Rows.Find(3);
 if( deleteRow != null )
 {
    deleteRow.Delete();           // delete the row
    dataTable.AcceptChanges();        // commit the changes
 }
```

Exception Handling;

- Exception Handling in C# is a process to handle runtime errors.
- Perform exception handling so that normal flow of the application can be maintained even after runtime errors.
- It is an event or object which is thrown at runtime.
- All exceptions are derived from System.Exception class.
- If we don't handle the exception, it prints exception message and terminates the program.

Exception	Description
System.DivideByZeroException	handles the error generated by dividing a number with zero.
System.NullReferenceException	handles the error generated by referencing the null object.
System.InvalidCastException	handles the error generated by invalid typecasting.
System.IO.IOException	handles the Input Output errors.
System.FieldAccessException	handles the error generated by invalid private or protected field access.

- use 4 keywords to perform exception handling:
  - try
  - catch
  - finally
  - throw

```
try
      {
         int a = 10;
         int b = 0;
         int x = a / b;
      }
catch (Exception e) { Console.WriteLine(e); }
finally { Console.WriteLine("Finally block is executed"); }

 if (age < 18)
{
   throw new InvalidAgeException("Sorry, Age must be greater than 18");
}

public class InvalidAgeException : Exception
{
   public InvalidAgeException(String message)
      : base(message)
   {

   }
}
```

Different Project types:

- **Console Application:**
  - **Description**: A simple application that runs in a console or command-line interface.
  - **Use Cases**: Basic utilities, small tools, or programs without a graphical user interface.
- **Windows Forms Application:**
  - **Description**: A desktop application with a graphical user interface (GUI) built using Windows Forms.
  - **Use Cases**: Traditional Windows desktop applications with UI elements like buttons, textboxes, and forms.
- **ASP.NET Web API:**
  - **Description**: A project for building HTTP-based services that can be consumed by web clients or mobile applications.
  - **Use Cases**: Building RESTful APIs for web and mobile applications.

Working with string class:

```
string name = "Dev Nakum";

// lenght of string
Console.WriteLine($"The lenght of name is {name.Length}");
```

```
// convert string into uppercase
Console.WriteLine($"Uppercase : {name.ToUpper()}");

// convert string into lowercase
Console.WriteLine($"Lowercase : {name.ToLower()}");

// remove the space in between word
Console.WriteLine($"Remnove the space : {name.Trim()}");

// find the index of first character
Console.WriteLine($"Index of N is: {name.IndexOf('N')}");

// find the index of first character
Console.WriteLine($"Last Index of N is : {name.LastIndexOf('Na')}");

// substring from index 2 to length of 4
Console.WriteLine($"Substring : {name.Substring(2,4)}");

// replace the word to another word
Console.WriteLine($"Replace Dev to Kishan: {name.Replace("Dev","Kishan")}");

// insert new word at specific index
Console.WriteLine($"Insert : {name.Insert(0,"Welcome ")}");

string stringNumber = "12312";
// convert string into int
int intNumber = int.Parse(stringNumber);
Console.WriteLine($"Convert string number into integer {intNumber}");

// convert int into string
Console.WriteLine($"Convert integer into string {intNumber.ToString()}");

// concatination
Console.WriteLine("Welcome "+name);

// comapre the string -- return 0 for same , -1 for assending, 1 from decending
Console.WriteLine($"Compare string {String.Compare("Dev" , "Kishan")}");

string[] stringArray = new string[]
{
   "Hello",
   "from",
   "another",
   "side",
```

```
};
// join with specific delimeter
string arrayToString = String.Join(" ", stringArray);
Console.WriteLine(arrayToString);

// comare the string -- return true or false
if (String.Equals("Dev", "Dev"))
{
   Console.WriteLine("Dev and Dev both are same");
}
else
{
   Console.WriteLine("Both are not same");
}
```

Working with DateTime Class:

```
DateTime dateTime = new DateTime(2024,01,05,17,51,56);
Console.WriteLine(dateTime);

// current date and time
Console.WriteLine(DateTime.Now);

// UTC date time
Console.WriteLine(DateTime.UtcNow);

//today's date and time
Console.WriteLine(DateTime.Today);

// calculate the days in specific month and year
Console.WriteLine(DateTime.DaysInMonth(1582, 10));

// add Days into current date
Console.WriteLine($"after 23 days date is {DateTime.Now.AddDays(18)}");

DateTime today = DateTime.Now;
Console.WriteLine($"ToLongDateString : {today.ToLongDateString()}");
Console.WriteLine($"ToShortDateString : {today.ToShortDateString()}");
Console.WriteLine($"ToLongTimeString : {today.ToLongTimeString()}");
Console.WriteLine($"ToShortTimeString : {today.ToShortTimeString()}");

// print into specific format
Console.WriteLine(today.ToString("dd/MM/yy"));
```

Basic File Operations:

```
string filePath = Path.Combine(Directory.GetCurrentDirectory(),@"..\..\data.txt");
string destinationFilePath = Path.Combine(Directory.GetCurrentDirectory(), @"..\..\data-copy.txt");


using (StreamWriter streamWriter = new StreamWriter(filePath))
 {
    streamWriter.WriteLine("Hello from data file !!!");
    streamWriter.WriteLine("This is a testing file for file handling.");
 }

// streamReader for read something from file
 using (StreamReader streamReader = new StreamReader(filePath))
 {
    // read and display each line from file
    while(!streamReader.EndOfStream)
    {
       string line = streamReader.ReadLine();
       Console.WriteLine(line);
    }
 }

FileInfo fileInfo = new FileInfo(filePath);

 // Display file operation
 Console.WriteLine($"File name : {fileInfo.Name}");
 Console.WriteLine($"File Directory Name : {fileInfo.Directory}");
 Console.WriteLine($"File Size: {fileInfo.Length} bytes");
 Console.WriteLine($"Creation Time: {fileInfo.CreationTime} bytes");
 Console.WriteLine($"Last Access Time: {fileInfo.LastAccessTime} bytes");
 Console.WriteLine($"Last Write Time: {fileInfo.LastWriteTime} bytes");

 // to check file is exists or not
 if(fileInfo.Exists)
 {
    Console.WriteLine("File is exists");

   // File.Copy(filePath, destinationFilePath);
    Console.WriteLine("Copied file successfully") ;
   // File.Delete(destinationFilePath);      // delete the file
    Console.WriteLine("Delete the file successfully") ;
 }
```