.NET

⊙ .NET

→ .NET is a software development framework created by Microsoft.

→ It provides a comprehensive and consistent programming model for building applications, including web, desktop and mobile applications.

→ It provides a runtime environment and set of libraries and tools for building and running applications on windows OS.

→ The framework include a ~~variety~~ variety of programming languages such as C#, F#, and Visual Basic and support ~~large number~~ a range of application types, including desktop, web, mobile, and gaming application.

⊙ Features of .NET

(1.) Multi Language support
→ allows developers to write code in various languages

(2.) Common Language runtime (CLR)
→ manages execution of .NET programs, including memory management and security.

(3.) Base Class Library (BCL)
→ provides a rich set of APIs and classes for common programming tasks, such as file I/O, data access, and user interface design.

(4.) Interoperability
→ support interaction with other languages and COM components.

(5.) Automatic Garbage Collection
→ Manages memory allocation and deallocation automatically.

(6.) Security
→ Provides robust security features including code access security and role based security.

(7.) Cross-language Integration.
→ Allows components written in different .NET languages to interact with each other.

(8.) Type Safety
→ ensures that your code only operates on the correct type of data. It helps prevent error by making sure you don't accidentally mix different types of data, such as trying to add a number to a text string.

⊙ Components of .NET

(1.) CLR (Common Language Runtime)
→ The execution engine that handles running application, memory management and garbage collection.

(2.) BCL (Base Class Library)

→ A comprehensive collection of classes and APIs that provide core functionalities

(3.) ASP.NET
→ A framework for building dynamic web applications and services.

(4.) Windows Forms
→ Provides a set of managed libraries for creating desktop application.

(5.) ADO.NET
→ A data access framework for interacting with databases

(6.) Entity Framework (EF)
→ An ORM (object relational mapper) that simplifies database operations and enables working with databases using .NET objects.

(7.) Xamarin
→ A framework for developing cross platform mobile applications for Android and iOS using .NET

(8.) WPF (Windows Presentation Foundation)
→ A UI framework for creating rich desktop application with advanced graphic and animation.

(9.) .NET Core / .NET 5+

→ A cross platform, open source version of .NET that runs on windows, macOS and Linux.

(10.) NuGet

→ A package manager for .NET that manages the installation and versioning of 3rd party libraries.

⊙ Advantages of .NET

↳ Cross Platform Development
↳ Language flexibility
↳ Rich class libraries
↳ Strong security
↳ Automatic Memory management
↳ Component based development
↳ Unified Development Model
↳ Active community and support

⊙ Disadvantages of .NET

↳ Platform Dependency (Older Versions → windows centric)
↳ Resource consumption
↳ Learning curve : can be complex for begineer
↳ Versioning Issues
↳ Framework size : The framework can be large, potentially affecting deployment and application size.

⊙ Applications of .NET

↳ Web Application
↳ Desktop Application
↳ Mobile Application
↳ Cloud Services
↳ Gaming
↳ Enterprise Solution
↳ IOT

⊙ MSIL (Microsoft Intermediate Language)

↳ It is a CPU-independent intermediate language that is generated by .NET compilers.
↳ It serves as a bridge between the high-level language code and the machine code.

How it works :

- compilation : when you compile .NET application, the high level code is first converted into MSIL, which is stored in an assembly (DLL or EXE file)
- JIT compilation : At runtime, the ~~.NET~~ MSIL is converted into native code by the Just-In-Time (JIT) compiler. This process ensure that the code can be executed on different hardware architectures.

⊙ CLR (Common Language Runtime)

↳ The CLR is the execution engine of the .NET Framework. It provides a runtime environment that handles the execution of .NET programs.

↳ convert program into native code.

↳ Responsibilites

- Memory Management : Automatic garbage collection and allocation of memory.
- Exception Handling : Provide a structured mechanism for handling runtime errors.
- Security : enforces code access security and provide a managed execution environment.
- JIT compilation : convert MSIL into native machine code at runtime.

↳ Example :

C# code

```
try {
    int [] number = new int[5];
    int number = number [10]; // This will
}                                cause an exception
catch (IndexOfRangeException ex)
{
    Console.WriteLine ("Exception caught :" +
                                    ex.Message);
}
```

⊙ CLS (Common Language Specification)

↳ is a set of rules and guidelines that .NET languages must follow to ensure interoperability and integration of code written in different languages.

↳ Features

- Common Language Features : Define a subset of a CTS that is ~~ass~~ accessible from all .NET language.
- Interoperability : Ensure that code written in one CLS-compliant language can be used by other CLS-compliant languages.

Eg. C# code

```
public class MyClass
  { public void MyMethod() { }
  }
```

This class can be ~~ac~~accessed from other .NET languages like VB.NET or F#.

⊙ CTS (Common Type System)

↳ The CTS define how types are declared, used, and managed in the runtime. It provides a framework for type safety and ~~interop~~ interoperability.

↳ Features

- Type Definitions : Defines both built-in types (eg. integer, strings) and user-defined types
- Type Safety : Ensure that types are used consistently across different languages.

eg.

```
int number = 42;
string text = "Hello";
```

⊙ Namespaces

→ are a way to organize code into logical groups, helping to avoid name conflicts and improve code readability.

→ Usage :

- Declaration : use the 'namespace' keyword to define a namespace.
- Access : Use the 'using' directive to include a namespace in your code.

→ Eg.

```
namespace MyApplication
{ class MyClass
  { public void MyMethod () { }
  }
}
using MyApplication;
```

```
class Program
{ static void main()
    { MyClass obj = new MyClass();
      obj.MyMethod();
    }
}
```

① Assemblies

→ are the building blocks of .NET application. They are compiled code libraries used for deployment, versioning and security.

→ Types:

- Executable (exe) : Assemblies that can be executed directly.
- Dynamic Link Library (DLL) : Assemblies that provide reusable code and components.

→ Features:

- Versioning : Allows multiple versions of the same assembly to coexist.
- Deployment : Simplifies deployment by bundling all necessary components.

Eg.

```
// In MyAssembly.cs
namespace MyLibrary
{ public class MyClass
    { public void MyMethod(){}
    }
}
```

⊙ Garbage Collection

→ It is the memory management feature of the CLR that automatically reclaims memory used by objects that are no longer in use.

→ Process:

- Mark and Sweep
  ↳ Identifies objects that are ~~not~~ no longer referenced and reclaims their memory.
- Generational Collection
  ↳ Organizes objects into generations to optimize collection performance.

→ Eg.

```
class Program
{ static void main()
  { MyClass obj = new MyClass();
    obj=null; // The object is eligible for garbage collection.
  }
  GC.Collect(); // Forcing garbage collection.
}
```

⊙ DDL Hell

→ The term "DLL Hell" refers to the problem that arise when dealing with different version of Dynamic Link Libraries (DLLs) in software application.

→ These issue often occur when multiple application or components require different versions of the same DLL, leading to conflicts and unpredictable behaviour.

→ The .NET framework provide mechanisms to address and resolve the DLL Hell issues through managed execution and a variety of features:

(1.) Assembly Versioning

- What It Means: In .NET assemblies (including DDLS) have ~~ver~~ version numbers that helps manage and differentiate b/t various version of the same assembly.
- How it Help: By specifying version numbers, .NET allows multiple version of an assembly to exist on the same system. Applications can specify which version of an assembly they need preventing conflict.

(2.) Global Assembly Cache (GAC)

- What It Means: The GAC is a machine wide store used to hold assemblies that are intended to be shared among multiple application.
- How It Helps: Assemblies installed in the GAC are accessible to all .NET application on the machine. The GAC handles versioning and ensure that the correct version of an assembly is used by application that require it.
- Eg. The 'System.Data.dll' assembly is often installed in the GAC. Multiple application can use it without requiring seperate copies.

(3) Assembly Binding Redirects

- What it Means: In .NET configuration file (like `app.config` or `web.config`), you can specify binding redirects to direct the application to use a different version of an assembly.
- How it Helps: This features allow application to use newer versions of assemblies while maintaining compatibility with older version, preventing issues caused by version mismatches.
- Eg. `configuration` in app.config.

```
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
        <assemblyIdentity name="MyLibrary" publicKeyToken=
            "abcdef123456" culture="neutral" />
        <bindingRedirect oldVersion="1.0.0.0-2.0.0.0"
            newVersion="2.0.0.0" />
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

→ This redirect request for any version b/t '1.0.0.0' & '2.0.0.0' to version '2.0.0.0'.

(4) Private Assemblies

- What It Means: Assemblies that are deployed with the application that uses them. Each application has its own private copy of these assemblies.
- How It Helps: Private assemblies eliminate conflicts b/t applications by keeping the DLLs seperate from each other. This avoids version conflict because each application has its own set of assemblies.

(5.) Side-by-Side Execution

- What It Means : .NET support the side by side execution of different versions of assemblies. Application can run concurrently with different version of the same assembly.
- How It Helps : This feature ensure that multiple applications running on the same machine can use different versions of the same DLL without interfering with each other.

(6.) Managed Execution

- What It Means : The .NET runtime (CLR) manages the execution of code ; including loading and binding assemblies and ensuring that the correct version of an assembly is used.
- How It Helps : By managing the execution environment, the CLR helps prevent conflicts and issues related to DLL versioning and deployment ensuring smooth execution and consistent behavior.

C#

→ is a modern, OOP language developed by Microsoft.
→ It is a part of the .NET Framework and is used for building wide range of application, from desktop to web to mobile

* Basic Syntax:

Write() → print the string provided to it
WriteLine() → print the string and moves to the start of next line as well

```
using System;
class Program
{ static void main()
    { console.writeLine ("Hello world");
    }
}
```

- `using System;` : Import the System namespace.
- `class Program` : Define a class named program.
- `static void Main()` : Entry point of the program.

* Variables & Data Types

```
int myInt = 10;                  // Integer
double myDouble = 5.99           // Floating-point number
char myChar = 'A';               // Character
string myString = "Hello"        // String
bool ~~oooooo~~ myBool = "True"  // Boolean
```

* Comments

1. Single Line Comments (//)
2. Multiline Comments (/* */)
3. XML Comments (///)

* Operators

```
int x=5;
int y=3;
int sum=x+y;    // Addition
int diff=x-y;   // Subtraction
int prod=x*y;   // Multiplication
int quot=x/y;   // Division
int mod=x%y;    // Modulus
```

* Control Structure

① Conditional Statements

```
int a=10;
if (a>5)
{ console.writeline("a is greater than 5");
}
else
{ console.writeLine("a is not greater than 5");
}
```

② Switch Statement

```
int day=2;
switch (day)
{ case 1:
        console.writeLine("Monday");
        break;
```

```
case 2 :
    Console.Writeline ("Tuesday");
    break;
default :
    Console.Writeline ("Other day");
    break;
}
```

(3) Loops

3.1 For Loop

```
for ( int i=0; i<5; i++)
  { console.Writeline (i);
    }
```

3.2 while Loop

```
int i=0;
while (i<5)
  { console.Writeline (i);
    i++;
  }
```

3.3 Do-while Loop

```
int i=0;
do
{ console.Writeline (i);
  i++;
} while (i<5);
```

* Functions.

⇒ Defining and calling Functions

```
class Program
{ static void Main()
  { int result = Add(5,3);
    Console.WriteLine(result);
  }

  static int Add (int a, int b)
  { return a+b;
  }
}
```

⇒ Function overloading

```
class Program
{ static void main ()
  { Console.WriteLine (Add (5,3));
    Console.WriteLine (Add (5.5,3.3));
  }

  static int Add (int a, int b)
  { return a+b;
  }

  static double Add (double a, double b)
  { return a+b;
  }
}
```

* Object - Oriented Programming (OOP)

① Classes and objects

```
class Person
{ public string Name;
  public int Age;
  public void Greet()
  { Console.WriteLine("Hello, my name is" + Name);
  }
}
class Program
{ static void Main()
  { Person p1 = new Person();
    p1.Name = "Kamlesh";
    p1.Age  = 21;
    p1.Greet();
  }
}
```

② Constructors

```
class Person
{ public string Name;
  public int Age;
  public Person(string name, int age)
  { Name = name;
    Age = age;
  }
```

```
    public void Greet()
    { Console.WriteLine("Hello, my name is" + Name);
    }
}
class Program
{ static void Main()
    { Person p1 = new Person("Kamlesh", 21);
      p1.Greet();
    }
}
```

⇒ Types of constructor

① Parameterless constructor
↳ constructor without parameters

② Parameterized constructor
↳ constructor can also accept parameters
↳ value passed to constructor are called parameters.

③ Default constructor
↳ If we not defined a constructor in our class, then the C# will automatically create a default constructor with an empty code and no parameter.

④ Copy Constructor
↳ we can use a copy constructor to create an object by copying data from another object.

⇒ Eg. of copy constructor

```
using System;
namespace Constructor {
    class Car {
        string brand;

        Car (string theBrand) {
            brand = theBrand;
        }
        // Copy Constructor
        Car (Car c1) {
            brand = c1.brand;
        }

        Static void Main (string[] args) {
            Car car1 = new Car ("Bugatti");
            Console.WriteLine ("Brand of car1 : " + car1.brand);

            Car car2 = new Car (car1);
            Console.WriteLine ("Brand of car2 : " + car2.brand);

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

③ Inheritance

→ In C#, we use : symbol to perform inheritance

```
class Animal
{ public void Eat()
  { console.WriteLine("Eating...");
  }
}
class Dog : Animal
{ public void Bark()
  { console.WriteLine("Barking...");
  }
}
class Program
{ static void main()
  { Dog dog = new Dog();
    dog.Eat();
    dog.Bark();
  }
}
```

⇒ Types of Inheritance

① Single Inheritance
② Multilevel Inheritance
③ Hierarchical Inheritance
④ Multiple Inheritance (C# doesn't support but we can achieve it through Interface)
⑤ Hybrid Inheritance

4. Polymorphism

→ means ~~occuring~~ occurring in more than one form

```
using System;
class Program
 { public void greet()
    { console.writeline("Hello");
    }
   public void greet(string name)
    { console.writeline("Hello " + name);
    }
   Static void main(string[] args)
    { Program p1 = new Program();
      p1.greet();
      p1.greet("Tim");
    }
 }
```

⇒ Types of Polymorphism

① Compile Time / Static Polymorphism

→ we can achieve through 2 ways

- method overloading
- Operator overloading

② Runtime / Dynamic Polymorphism

- Method overriding

Method overriding

→ we can use virtual and override keyword to achieve method overriding

```
using System;
class Polygon
{ public virtual void render()
    { console.writeline("Rendering Polygon..");
    }
}
class Square : Polygon
{ public override void render()
    { console.writeline("Rendering Square..");
    }
}
class myProgram
{ public static void main()
    { Polygon obj1 = new Polygon();
      obj1.render();
      obj1 = new Square();
      obj1.render();
    }
}
```

O/P: Rendering Polygon..
Rendering Square..

Note: Virtual → allows the method to be overridden by the derived class
override → indicate the method is overriding the method from the base class.

(5) Encapsulation

```
class Person
{ private string name;
  public string Name
    { get { return name; }
      set { name = value; }
    }
}
class Program
{ static void Main()
    { Person p1 = new Person();
      p1.Name = "Kamleshi";
      Console.WriteLine(person.Name);
    }
}
```

(6) Interface

→ All the methods of interface are fully abstract (method without body)
→ we use interface keyword to create interface
→ We cannot create objects of an interface
→ To use an interface, other classes must implement it
→ Same as in C# Inheritance, we use : symbol to implement an interface.
→ Interface start with I so that we can identify it just by seeing its name.
→ a class can implement multiple interface.

```
using System;
namespace CsharpInterface{

    interface IPolygon{
        void // method without body
        void calculateArea(int l, int b);
    }

    class Rectange : IPolygon{

        //implementation of method inside interface
        public void calculateArea(int l, in b){
            int area = l * b;
            Console.WriteLine("Area of Rectangle:"+area);
        }
    }

    class Program{
        Static void Main(string[] args){
            Rectangle r1 = new Rectangle();
            r1.calculateArea(100, 200);
        }
    }
}
```

Note: We must provide the implementation of all method of interface inside the class that implements it.

(7) Abstract Classes

→ We cannot create ~~object~~ object of abstract class
→ We use abstract keyword to create an abstract class
→ An abstract class can have both abstract method (method without body) and non-abstract method (method with the body).
→ As we cannot create object of an abstract class, we must create a derived class from it, so that we can access members of the abstract class using the object of the derived class.

```
abstract class Animal
 { public abstract void MakeSound();
    public void sleep()
      { console.WriteLine("Sleeping...");
      }
 }
class Dog : Animal
 { public override void MakeSound()
     { console.WriteLine("Barking...");
     }
 }
class Program
{ static void Main()
   { Dog dog = new Dog();
     dog.MakeSound;
     dog.Sleep();
   }
}
```

* Exception Handling

```
class Program
{ static void Main()
  { try
    { int [] number = {1,2,3};
      Console.WriteLine (number [3]);
    }
    catch (IndexOutOfRangeException ex)
    { Console.WriteLine ("Index out of range:" +
                         ex.Message);
    }
    finally
    { Console.WriteLine ("This will always run");
    }
  }
}
```

* Collections

(#) ~~Arrays~~

⇒ In C# collections are divided into 3 classes.

(1.) System.Collection.Generic
(2.) System.Collection
(3.) System.Collection.Concurrent

(1) System.Collections.Generic
→ help us to create generic collection
→ we store type compatible data elements
→ does not allow us to store different types of element
→ In C#, following are the classes that comes under System.Collections.Generic namespace.
(1.) List Class
(2.) Stack Class
(3.) Queue Class
(4.) SortedList Class

(2) System.Collections
→ In C#, the system.collection classes help us to create non-generic collection.
→ Using this we can create ~~&~~ classes where we can add data elements of multiple datatype
→ Following are some of the classes that are in System.Collections namespace:
(1.) ArrayList Class
(2.) Hashtable Class

(3) System.Collection.Concurrent
→ provide some collection classes that help to achieve thread - ~~same~~ safe code.

What is Thread safety?

→ There can be situation when multiple thread are trying to execute the same piece of code. The code is said to be "thread-safe" if it can be executed correctly irrespective of multiple thread accessing concurrently.

→ Some of the thread safe collection classes are:

- ConcurrentStack<T>
- ConcurrentQueue<T>
- ConcurrentDictionary<TKey, TValue>