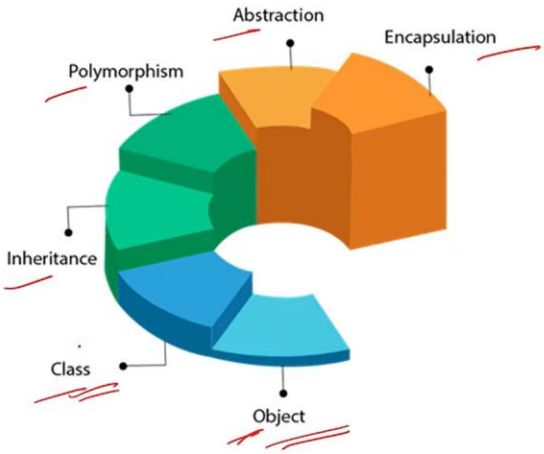
Q

What is OOPs and What are the main concepts of OOPs? V. IMP.

OOP

- OOP stands for **Object-Oriented Programming**, which means it is way to create software around **objects**.
- OOPs provide a clear structure for the software's and web applications.

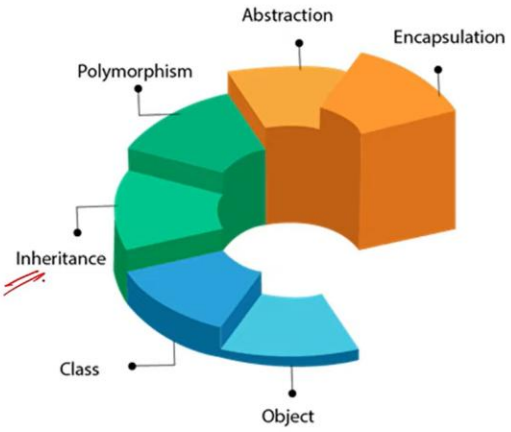


Q What are the advantages and limitations of OOPS?

OOP

❖ Advantages of OOPS:

1. **Reuse** of code using inheritance.
2. **Flexibility** of code using polymorphism.
3. **Secure** application by using Encapsulation.
4. **Easily scalable** from small to large applications.
5. **Easier troubleshooting** of code because of modularity.



❖ Disadvantage of OOPS:

1. It is not suitable for **small** applications.

Q What are Classes and Objects?

OOP

❖ A class is a **LOGICAL UNIT or BLUEPRINT.** It contains fields, methods and properties.

❖ Class members are:

1. A **constructor** is a method in the class which gets executed when a class object is created.

2. A **field** is a variable of any type. It is basically the data.

3. A **property** is a member that provides helps in read and write of private field.

4. A **method** is a code block that contains a series of statements.

```
public class Employee
{
    public Employee()
    {
        //code
    }

    private int experience;

    public int Experience
    {
        get { return experience; }

        set { experience = value; }
    }

    //public int Experience { get; set; }

    public void CalculateSalary()
    {
        int salary = Experience * 300000;

        Console.WriteLine(salary);
    }
}
```

Constructor

Field

Property

Method

What are Classes and Objects?

OOP

- **Object** - An object is an **INSTANCE** of a class.

```
static void Main(string[] args)
{
    Employee objEmployee = new Employee();

    objEmployee.Experience = 3;

    objEmployee.CalculateSalary();

    Console.ReadLine();
}
```

What is Inheritance? Why Inheritance is important?

OOP

- Inheritance is creating a **PARENT-CHILD** relationship between two classes, where child class will **automatically** get the properties and methods of the parent.

- Inheritance is good for: **REUSABILITY** and **ABSTRACTION** of code

```
public class ContractEmployee : Employee
{
    //No method or Property here
}
```

```
public class Employee
{
    public int Experience { get; set; }

    public void CalculateSalary()
    {
        int salary = Experience * 300000;

        Console.WriteLine("salary:{0} ", salary);
    }
}
public class PermanentEmployee : Employee
{
    //No method or Property here
}

static void Main(string[] args)
{
    PermanentEmployee pEmployee = new PermanentEmployee();
    pEmployee.Experience = 5;
    pEmployee.CalculateSalary();

    Console.ReadLine();
}
```

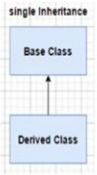
Base/ Parent/ Super class

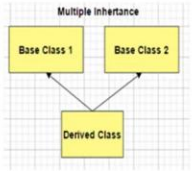
Derived/ Child/ Sub class

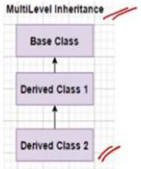
CalculateSalary() method is not present in PermanentEmployee class, but it will get it automatically from it's parent

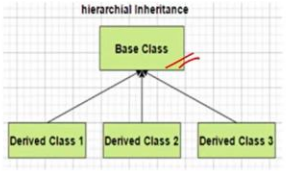
Q What are the different types of Inheritance? V. IMP.

OOP










```
class BaseClass1
{
    public void Animal()
    {
        Console.WriteLine("Animal");
    }
}
class DerivedClass1 : BaseClass1
{
    public void Dog()
    {
        Console.WriteLine("Dog");
    }
}
```

```
class BaseClass2
{
    public void Animal()
    {
        Console.WriteLine("Animal");
    }
}
interface I2
{
    public void Fly();
}
class DerivedClass2 : BaseClass2, I2
{
    public void Eagle()
    {
        Console.WriteLine("Eagle");
    }
    public void Fly()
    {
        Console.WriteLine("Fly");
    }
}
```

```
class BaseClass3
{
    public void Animal()
    {
        Console.WriteLine("Animal");
    }
}
class DerivedClass3 : BaseClass2
{
    public void Dog()
    {
        Console.WriteLine("Dog");
    }
}
class DerivedClass4 : DerivedClass3
{
    public void Labrador()
    {
        Console.WriteLine("Labrador");
    }
}
```

```
class BaseClass4
{
    public void Animal()
    {
        Console.WriteLine("Animal");
    }
}

class DerivedClass5 : BaseClass4
{
    public void Dog()
    {
        Console.WriteLine("Dog");
    }
}
class DerivedClass6 : BaseClass4
{
    public void Cat()
    {
        Console.WriteLine("Dog");
    }
}
```

Q How to prevent a class from being Inherited?

OOP

- By using **SEALED** keyword in class

```
public sealed class Employee
{
    public int Experience { get; set; }

    public void CalculateSalary()
    {
        int salary = Experience * 300000;

        Console.WriteLine("salary:{0} ", salary);
    }
}

public class PermanentEmployee : Employee
{

}
```

class How_to_prevent_a_class_from_being_Inherited.Employee

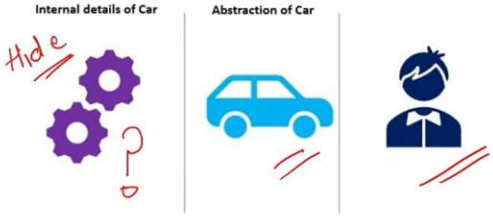
CS0509: 'PermanentEmployee': cannot derive from sealed type 'Employee'

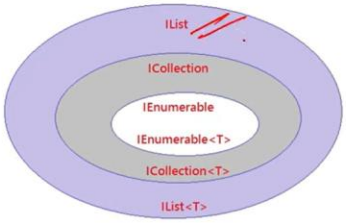
Show potential fixes (Alt+Enter or Ctrl+.)

What is Abstraction?

OOP

- Abstraction means showing only required things and hide the **BACKGROUND** details.






```
static void Main(string[] args)
{
    String name = "InterviewHappy";

    name.Substring(8);

    Console.WriteLine(name);

    Console.ReadLine();
}
```

Background/Hidden methods. Developer not aware about it.

```
public abstract class EmployeeSalary
{
    public int Experience { get; set; }

    public void CalculateSalary()
    {
        int salary = Experience * 300000;
    }

    public abstract void CalculateBonus();
}
```

What is Encapsulation?

OOP

- Encapsulation means **WRAPPING** of data and methods/properties into a single unit.

```
public class Employee
{
    public int empExperience;
}

static void Main(string[] args)
{
    Employee objEmployee = new Employee();

    objEmployee.empExperience = 3;
}
```

Field/Data is accessed without property or function. It will work but it violating encapsulation

```
public class Employee
{
    //Make field private
    private int empExperience;

    public int EmpExperience
    {
        get { return empExperience; }

        set { empExperience = value; }
    }

    //Shortcut Property
    //public int EmpExperience { get; set; }
}
```

This field cannot be accessed from outside without the property

What is Encapsulation?

OOP

- Encapsulation means **WRAPPING** of data and methods/properties into a single unit.

```
public class Employee
{
    public int empExperience;
}

static void Main(string[] args)
{
    Employee objEmployee = new Employee();

    objEmployee.empExperience = 3;
}
```

Field/Data is accessed without property or function. It will work but it violating encapsulation

```
public class Employee
{
    //Make field private
    private int empExperience;

    public int EmpExperience
    {
        get { return empExperience; }

        set { empExperience = value; }
    }

    //Shortcut Property
    //public int EmpExperience { get; set; }
}
```

This field cannot be accessed from outside without the property

```
static void Main(string[] args)
{
    Employee objEmployee = new Employee();

    objEmployee.EmpExperience = 3;
}
```

What is Polymorphism and what are its types?

OOP

- Polymorphism is the ability of a variable, object, or function to take on **MULTIPLE FORMS**.

For example, in English "RUNNING" word can be used for "running a race" or "running a business". In both cases the meaning is different.


```
public class Polymorphism
{
    public int Add(int a, int b)
    {
        return a + b;
    }

    public string Add(string str1, string str2)
    {
        return str1 + str2;
    }
}
```

Function with same name but have multiple forms

```
static void Main(string[] args)
{
    Polymorphism obj = new Polymorphism();

    int i = obj.Add(50, 60);

    string str = obj.Add("Interview", "Happy");

    Console.WriteLine(i + " - " + str);

    Console.ReadLine();
}
```

Q What is Method Overloading? In how many ways a method can be overloaded? **V. IMP.**

OOP

- Method overloading is a type of polymorphism in which we can create multiple methods of the **same name in the same class**, and all methods work in different ways.

- It's a compile time polymorphism.

```
public class MethodOverloading
{
    public int Add(int a, int b)
    {
        return a + b;
    }

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

    public double Add(double a, double b, int c)
    {
        return a + b + c;
    }

    public double Add(double a, int c, double b)
    {
        return a + b + c;
    }
}
```

1. Number of parameters are different

2. Type of parameters are different

3. Order of parameters is different

Q What is the difference between Overloading and Overriding? V. IMP.

OOP

❖ Method Overloading

1. Multiple methods of same name in **single class**.
2. No need of **inheritance**, as it is in single class.
3. All methods have **different signature.**
4. It's a **compile time** polymorphism.
5. No special keyword used.

```
public class MethodOverloading
{
    public int Add(int a, int b)
    {
        return a + b;
    }

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

    public double Add(double a, double b, int c)
    {
        return a + b + c;
    }

    public double Add(double a, int c, double b)
    {
        return a + b + c;
    }
}
```

Q What is the difference between Overloading and Overriding? V. IMP.

OOP

❖ Method Overriding

1. Multiple methods of same name in **different class**.
2. **Inheritance is used**, as it is in different class.
3. All methods have **same signature.**
4. It's a **run time** polymorphism.
5. **Virtual** & **override** keywords.

```
public class BaseClass
{
    public virtual void Greetings()
    {
        Console.WriteLine("BaseClass Hello!");
    }
}

public class DerivedClass : BaseClass
{
    public override void Greetings()
    {
        Console.WriteLine("DerivedClass Hello!");
    }
}

static void Main(string[] args)
{
    DerivedClass objDerived = new DerivedClass();
    objDerived.Greetings();
    Console.ReadLine();
}

//Ouptut: DerivedClass Hello
```

Q What is the difference between Overloading and Overriding? V. IMP.

OOP

❖ Method Overloading

1. Multiple methods of same name in **single class**.
2. No need of **inheritance**, as it is in single class.
3. All methods have **different signature.**
4. It's a **compile time** polymorphism.
5. No special keyword used.

❖ Method Overriding

1. Multiple methods of same name in **different class**.
2. **Inheritance is used**, as it is in different class.
3. All methods have **same signature.**
4. It's a **run time** polymorphism.
5. **Virtual** & **override** keywords.

Q What is the difference between Method Overriding and Method Hiding?

OOP

❖ In Method Hiding, you can completely hide the implementation of the methods of a base class from the derived class using the **new keyword**.

```
public class BaseClass
{
    public virtual void Greetings()
    {
        Console.WriteLine("BaseClass Hello!");
    }
}

public class DerivedClass : BaseClass
{
    public override void Greetings()
    {
        Console.WriteLine("DerivedClass Hello!");
    }
}

static void Main(string[] args)
{
    BaseClass objDerived = new DerivedClass();

    objDerived.Greetings();
    Console.ReadLine();
}

//Ouptut: DerivedClass Hello
```

```
public class BaseClass
{
    public void Greetings()
    {
        Console.WriteLine("BaseClass Hello!");
    }
}

public class DerivedClass : BaseClass
{
    public new void Greetings()
    {
        Console.WriteLine("DerivedClass Hello!");
    }
}

static void Main(string[] args)
{
    BaseClass objDerived = new DerivedClass();

    objDerived.Greetings();
    Console.ReadLine();
}

//Ouptut: BaseClass Hello
```

Q What is the difference between an Abstract class and an Interface (atleast 4)? **V. IMP.** OOP

1. Abstract class contains both **DECLARATION & DEFINITION** of methods.

```
public abstract class Employee
{
    public abstract void Project();

    public void Role()
    {
        Console.WriteLine("Engineer");
    }
}
```

Method Declared

Method Defined

1. Interface should contain **DECLARATION** of methods.

❖ *With C# 8.0, you can now have default implementations/ definition of methods in an interface. But that is recommended in special case*.*

```
interface IEmployee
{
    public void Project1();

    public void Manager1();
}
```

Only method Declaration is allowed

2. Abstract class keyword: **ABSTRACT**

2. Interface keyword: **INTERFACE**

Q What is the difference between an Abstract class and an Interface (atleast 4)? **V. IMP.** OOP

3. Abstract class does not support **multiple inheritance**

```
public abstract class Employee
{
    public abstract void Project();

    public void Role()
    {
        Console.WriteLine("Engineer");
    }
}

public abstract class Employee1
{
    public abstract void Project1();

    public void Role1()
    {
        Console.WriteLine("Engineer1");
    }
}

public class PermanentEmployee: Employee, Employee1
{

}
```

3. Interface supports **multiple inheritance.**

```
interface IEmployee1
{
    public void Project1();
}

interface IEmployee2
{
    public void Project2();
}


public class NewEmployee : IEmployee1, IEmployee2
{
    public void Project1()
    {
        Console.WriteLine("Print 1");
    }

    public void Project2()
    {
        Console.WriteLine("Print 2");
    }
}
```

Q What is the difference between an Abstract class and an Interface (atleast 4)? V. IMP.

OOP

4. Abstract class can have **constructors**.

```
public abstract class Employee1
{
    public Employee1()
    {

    }

    public abstract void Project1();

    public void Role1()
    {
        Console.WriteLine("Engineer1");
    }
}
```

4. Interface do not have constructors.

```
interface IEmployee1
{
    public IEmployee1()
    {

    }
    public void Project1();
}
```

Q What is the difference between an Abstract class and an Interface (atleast 4)? **V. IMP.**

OOP

Abstract Class	Interface
1. Abstract class contains both **DECLARATION & DEFINITION** of methods.	Mostly Interfaces contain **DECLARATION** of methods. From C# 8.0 definition is also possible.
2. Abstract class keyword: **ABSTRACT**	2. Interface keyword: **INTERFACE**
3. Abstract class does not support **multiple inheritance**	3. Interface supports **multiple inheritance.**
4. Abstract class can have **constructors**.	4. Interface do not have constructors.

Q When to use Interface and when Abstract class?

OOP

❖ When to use Interface?

An interface is a good choice when you know a method has to be there, but it can be implemented **DIFFERENTLY** by independent derived classes.

```
public class PermanentEmployee
{

}

public class ContractualEmployee
{

}
```

```
interface IEmployee
{
    public void AssignEmail();

    public void AssignManager();
}
```

Q When to use Interface and when Abstract class?

OOP

- When to use Abstract class?

Abstract class is a good choice when you are sure some methods are concrete/defined and must be implemented in the **SAME WAY** in all derived classes.

- Normally we prefer Interface because it gives us the flexibility to modify the behavior at later stage.

```
public class PermanentEmployee
{

}

public class ContractualEmployee
{

}
```

```
public abstract class EmployeeDress
{
    public abstract void DressCode();

    public void DressColor()
    {
        Console.WriteLine("BLUE");
    }
}
```

Q Why to even create Interfaces?

OOP

- Benefits of Interfaces:

1. Help in defining the **contract** or boundaries of the system.
2. **Unit testing** is easy in application having interfaces.
3. Used for implementing dependency injection.

```
public class PermanentEmployee
{

}

public class ContractualEmployee
{

}

public class TemporaryEmployee
{

}
```

```
interface IEmployee
{
    public void AssignEmail();

    public void AssignManager();
}
```

Q Do Interface can have a Constructor?

OOP

- **NO**. Interface can only be used for inheritance, so constructor is not required.

Q Can an Interface have private access specifier?

OOP

- **NO**. Interface cannot be private, protected, protected internal. It can be public or internal only.

Q Can you create an instance of an Abstract class or an Interface?

OOP

- **NO**. Abstract class and Interface can only used for inheritance not for object creation.

```
static void Main(string[] args)
{
    IEmployee iemployee = new IEmployee();

    Employee employee = new Employee();
}
```

```
interface IEmployee
{
    public void Salary();

}

public abstract class Employee
{
    public abstract void Project();

    public void Role()
    {
        Console.WriteLine("Engineer");
    }
}
```

Q What are Access Specifiers? What is the default access modifier in a class?

OOP

- Access specifiers are keywords to specify the accessibility of a class, method, property, field.
- The keywords are – Public, Private, Protected, Internal, Protected Internal.

A1 A2

Access Specifier	Inside Same Assembly where member is declared			Other Assembly where containing Assembly is referenced	
	Inside Same Class	Inside Derived Class	Other Code	Inside Derived Class	Other Code
Public	✓	✓	✓	✓	✓
Private	✓	✗	✗	✗	✗
Internal	✓	✓	✓	✗	✗
Protected	✓	✓	✗	✓	✗
ProtectedInternal	✓	✓	✓	✓	✗

Q What are Access Specifiers? What is the default access modifier in a class?

OOP

- Access specifiers are keywords to specify the accessibility of a class, method, property, field.
- The keywords are – Public, Private, Protected, Internal, Protected Internal.

Access Specifier	Inside Same Assembly where member is declared			Other Assembly where containing Assembly is referenced	
	Inside Same Class	Inside Derived Class	Other Code	Inside Derived Class	Other Code
Public	✓	✓	✓	✓	✓
Private	✓	✗	✗	✗	✗
Internal	✓	✓	✓	✗	✗
Protected	✓	✓	✗	✓	✗
ProtectedInternal	✓	✓	✓	✓	✗

Internal is the default access modifier of a class.

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

Q What is Boxing and Unboxing? Where we use them? V. IMP.

OOP

- **Boxing -** Boxing is the process of converting from value type to reference type.
- **Unboxing -** Unboxing is the process of converting reference type to value type.

  Unboxing is explicit conversion process.

```
static void Main(string[] args)
{
    int num = 100;

    object obj = num;      //Boxing

    int i = (int)obj;      //Unboxing
}
```

- We internally use boxing when item is added to Arraylist. And we use unboxing when item is extracted from Arraylist.

```
ArrayList arrayList = new ArrayList();

arrayList.Add(i);

int k = (int)arrayList[0];

Console.WriteLine(k);
```

Q What is the difference between "String" and "StringBuilder"? When to use what?

OOP

- String is **IMMUTABLE** in C#.

It means if you defined one string then you couldn't modify it. Every time you will assign some value to it, it will create a new string.

```
String str1 = "Interview";

str1 = str1 + "Happy";

Console.WriteLine(str1);
```

Both these strings are different and occupy different memory in process

- StringBuilder is **MUTABLE** in C#.

This means that if any manipulation will be done on string, then it will not create a new instance every time.

```
StringBuilder str2 = new StringBuilder();

str2.Append("Interview");

str2.Append("Happy");
```

Both these strings will remain same in memory.

- If you want to change a string multiple times, then StringBuilder is a better option from **performance** point of view because it will not require new memory every time.

Q What are the basic string operations in C#?

OOP

- **Concatenate:**
  Two strings can be concatenated either by using a System.String.Concat or by using + operator.

```
string str1 = "This is one";
string str2 = "This is two";
string str2 = str1 + str2;
//Output: This is one This is two
```

- **Replace:**
  Replace(a,b) is used to replace a string with another string.

```
string str1 = "This is one";
string str2 = str1.Replace("one", "two");
//Output: This is two
```

- **Trim:**
  Trim() is used to trim the white spaces in a string at the end.

```
string str1 = "This is one  ";
str1.Trim();
//Output: "This is one"
```

- **Contains:**
  Check if a string contains a pattern of substring or not.

```
string str = "This is test";
if (str.Contains("test"))
{
    Console.WriteLine("The 'test' was found.");
}
//Output: The 'test' was found.
```

Q What are Nullable types?

OOP

- Nullable types - To **hold the null values** we have to use nullable types because variable types does not hold null values.

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

    int i = null;

    //  Valid declaration
    Nullable<int> j = null;

    // Valid declaration
    int? k = null;

}
```

Q Explain Generics in C#? When and why to use them? V. IMP.

OOP

- Generics allows us to make classes and methods - **type independent or type safe.**

```
static void Main(string[] args)
{
    bool equal = Calculator.AreEqual(4, 4);
bool strEqual = Calculator.AreEqual("Interview", "Happy");
}
```

```
public class Calculator
{
    public static bool AreEqual(int value1, int value2)
    {
        return value1.Equals(value2);
    }
}
```

- The problem is, it involves Boxing from converting int (value) to object (reference) type. This will impact the performance.

```
public static bool AreEqual(object value1, object value2)
{
    return value1.Equals(value2);
}
```

Q Explain Generics in C#? When and why to use them?

OOP

- Generics allows us to make classes and methods - **type independent or type safe**.

- Generic Method

```
static void Main(string[] args)
{
    //bool equal = Calculator.AreEqual(4, 4);
    //bool strEqual = Calculator.AreEqual("Interview", "Happy");

    bool equal = Calculator.AreEqual<int>(4, 4);

    bool strEqual = Calculator.AreEqual<string>("Interview", "Happy");
}
```

```
public class Calculator
{
    public static bool AreEqual<T>(T value1, T value2)
    {
        return value1.Equals(value2);
    }
}
```

Q Explain Generics in C#? When and why to use them?

OOP

- Generics allows us to make classes and methods - **type independent or type safe**.

- Generic Class

```
static void Main(string[] args)
{
    //bool equal = Calculator.AreEqual(4, 4);
    //bool strEqual = Calculator.AreEqual("Interview", "Happy");

    bool equal = Calculator<int>.AreEqual(4, 4);

    bool strEqual = Calculator<string>.AreEqual("Interview", "Happy");
}
```

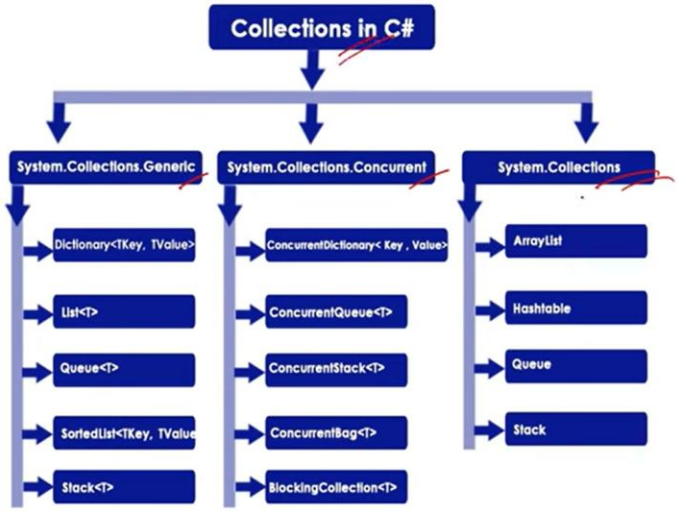
```
public class Calculator<T>
{
    public static bool AreEqual(T value1, T value2)
    {
        return value1.Equals(value2);
    }
}
```

Q

What are Collections in C# and what are their types?

OOP

- C# collection are designed to **store, manage and manipulate** similar DATA more efficiently.
- *For example ArrayList, Dictionary, List, Hashtable etc.*
- *Data manipulation meaning adding, removing, finding, and inserting data in the collection.*



Q How to implement Exception Handling in C#?

OOP

- Exception handling in Object-Oriented Programming is used to **MANAGE ERRORS.**

1. **TRY** – A try block is a block of code inside which any error can occur.
2. **CATCH** – When any error occur in TRY block then it is passed to catch block to handle it.
3. **FINALLY** – The finally block is used to execute a given set of statements, whether an exception is thrown or not thrown.

```
try
{
    int i = 1;
    int j = 1;

    int k = i / j;
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);

    throw;
}
finally
{
    Console.WriteLine("Finally");
    Console.ReadLine();
}
```

Q Can we execute multiple Catch blocks?

OOP

- **NO**

*We can write multiple catch blocks but when we will run the application and if any error occur, **only one** out of them will execute based on the type of error.*

```
static void Main(string[] args)
{
    try
    {
        int i = 0;
        int j = 0;

        int k = i / j;
    }
    catch (ArithmeticException ex)
    {
        Console.WriteLine(ex.Message);
    }
    catch (ArgumentOutOfRangeException ex)
    {
        Console.WriteLine(ex.Message);
    }
}
```

Q What is a Finally block and give an example when to use it?

OOP

- Finally block will be executed IRRESPECTIVE of exception.

```
static void Main(string[] args)
{
    SqlConnection con = new SqlConnection("conString");
    try
    {
        con.Open();

        //Some logic

        // Error occurred

        con.Close();
    }
    catch(Exception ex)
    {
        // error handled
    }
    finally
    {
        // Connection closed
        con.Close();
    }
}
```

Q Can we have only "Try" block without "Catch" block?

OOP

- YES - We can have only try block without catch block but we have to have **finally** block then.

Finally, will execute even this function return from here.

```
static void Main(string[] args)
{
    SqlConnection con = new SqlConnection("conString");
    try
    {
        con.Open();
        Random rnd = new Random();
        int num = rnd.Next();

        if(num == 5)
        {
            return;
            //After this nothing will execute
        }
        //con.Close();
    }
    finally
    {
        con.Close();
    }
}
```

Q What is the difference between "throw ex" and "throw"? **V. IMP.**

OOP

- Throw ex will change the stack trace, where as throw will preserve the whole **stack trace.**

Error:

at Throw_ex__and__throw.Program.DivideZerobyZero() in D:\InterviewHappy_InterviewHappy-Hindi\31-Oct-2022\OOP_CSharp_Code\Throw_ex_Throw\Program.cs:line 32
at Throw_ex__and__throw.Program.Main(String[] args) in D:\InterviewHappy_InterviewHappy-Hindi\31-Oct-2022\OOP_CSharp_Code\Throw_ex_Throw\Program.cs:line 15

```
11      static void Main(string[] args)
12      {
13          try
14          {
15              DivideZerobyZero();
16          }
17          catch (Exception ex)
18          {
19              Console.WriteLine(ex.StackTrace);
20              Console.ReadLine();
21          }
22      }
23      public static void DivideZerobyZero()
24      {
25          try
26          {
27              int i = 0, j = 0;
28              int k = i / j;
29          }
30          catch (Exception ex)
31          {
32              throw ex;
33          }
34      }
```

Q What is the difference between "throw ex" and "throw"? **V. IMP.**

OOP

- Throw ex will change the stack trace, where as throw will preserve the whole stack trace.

Error:

at Throw_ex__and__throw.Program.DivideZerobyZero() in D:\InterviewHappy_InterviewHappy-Hindi\31-Oct-2022\OOP_CSharp_Code\Throw_ex_Throw\Program.cs:line 28
at Throw_ex__and__throw.Program.Main(String[] args) in D:\InterviewHappy_InterviewHappy-Hindi\31-Oct-2022\OOP_CSharp_Code\Throw_ex_Throw\Program.cs:line 15

- Its a best practice to use *throw* as it preserve the whole **stack trace.**

```
11      static void Main(string[] args)
12      {
13          try
14          {
15              DivideZerobyZero();
16          }
17          catch (Exception ex)
18          {
19              Console.WriteLine(ex.StackTrace);
20              Console.ReadLine();
21          }
22      }
23      public static void DivideZerobyZero()
24      {
25          try
26          {
27              int i = 0, j = 0;
28              int k = i / j;
29          }
30          catch (Exception ex)
31          {
32              throw;
33          }
34      }
```

Q What are the Loop types in C#?

OOP

```
//While Loop

int i = 0;
while(i < 5)
{
    Console.WriteLine(i);
    i++;
}

//Output: 0 1 2 3 4
```

Initialize

Condition

Increment

```
//Do While

int j = 100;
do
{
    Console.WriteLine(j);
    j++;
}
while(j < 10);

//Output: 100
```

Whether condition true or false, this statement will run at least first time.

```
//For Loop

for(int k = 0 ; k < 5; k++)
{

    Console.WriteLine(k);

}

//Output: 0 1 2 3 4
```

```
//ForEach Loop

int[] arr = new int[] { 1,2,3,4};

foreach (int items in arr)
{
    Console.WriteLine(items);
}

//Output: 1 2 3 4
```

Q What is the difference between "continue" and "break" statement?

OOP

- **Continue** statement is used to skip the remaining statements inside the loop and transfers the control to the **beginning** of the loop.

```
static void Main(string[] args)
{
    for (int i = 0; i < 5; i++)
    {
        if (i == 3)
        {
            continue;
        }
        Console.WriteLine("Print: " + i);
    }
    Console.ReadLine();
}
```

```
//Output:
Print: 0
Print: 1
Print: 2
Print: 4
```

- **Break** statement breaks the loop. It makes the control of the program to **exit** the loop.

```
static void Main(string[] args)
{
    for (int i = 0; i < 5; i++)
    {
        if (i == 3)
        {
            break;
        }
        Console.WriteLine("Print: " + i);
    }
    Console.ReadLine();
}
```

```
//Output:
Print: 0
Print: 1
Print: 2
```

Q What is the difference between Array and ArrayList (atleast 2)?

OOP

- Array

1. Array is STRONGLY typed. This means that an array can store only specific type of items/ elements.

```
static void Main(string[] args)
{
    int[] array;

    array = new int[10];

    array[0] = 1;

    array[1] = "Happy";
}
```

2. Array can contain FIXED number of items.

- ArrayList

1. ArrayList can store ANY type of items\elements.

```
static void Main(string[] args)
{
    ArrayList arrayList;
    arrayList = new ArrayList();

    arrayList.Add(1);
    arrayList.Add("Happy");
}
```

2. ArrayList can store ANY number of items.

Q What is the difference between Arraylist and Hashtable? V. IMP.

OOP

- ArrayList - In Arraylist we can only add Items/ Values to the list.

```
ArrayList arrList = new ArrayList();

arrList.Add(7896);

arrList.Add("Happy");
```

Value

- Hashtable - . In Hashtable we can add Items/Values with the Keys.

```
Hashtable hashTable = new Hashtable();

hashTable.Add("Number", 1);

hashTable.Add("Car", "Ferrari");
```

Key Value

Q What is IEnumerable in C#? V. IMP.

OOP

- IEnumerable interface is used when we want to ITERATE among our collection classes using a **FOREACH** loop.

```
static void Main(string[] args)
{
    var employees = new List<Employee>() {
        new Employee(){ Id = 1, Name="Bill" },
        new Employee(){ Id = 2, Name="Steve" }
    };

    foreach (var employee in employees)
    {
        Console.WriteLine(employee.Id + ", " + employee.Name);
    }

    Console.ReadLine();
}

public class Employee
{
    public int Id { get; set; }
    public string Name { get; set; }
}
```

Q What is IEnumerable in C#? V. IMP.

OOP

```
namespace System.Collections.Generic
{
    public class List<T> : ICollection<T>, IEnumerable<T>, IEnumerable, IList<T>,
    {
        public List();
        public List(IEnumerable<T> collection);
        public List(int capacity);
```

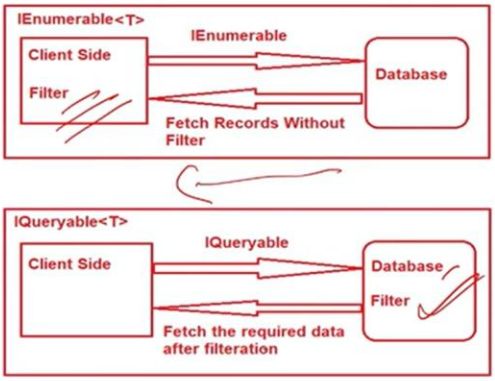
```
public interface IEnumerable
{
    //
    // Summary:
    //     Returns an enumerator that iterates through a collection.
    //
    // Returns:
    //     An System.Collections.IEnumerator object that can be used to iterate through
    //     the collection.
    IEnumerator GetEnumerator();
}
```

Q What is the difference between IEnumerable and IQueryable in C#? Why to use IQueryable in sql queries?

OOP

- IQueryable is also like IEnumerable and it is used to iterate sql query collection from data.

  It is under SYSTEM.LINQ namespace.




```
DataClassesDataContext dc = new DataClassesDataContext();
IQueryable<Employee> list = dc.Employees.Where(d => d.Department.Equals("IT"));
list = list.Take<Employee>(3);

foreach (var a in list)
{
    Response.Write(a.Name+"<br/>");
}
```

Q What is the difference between "out" and "ref" parameters?

OOP

- By using ref and out keywords we can pass parameters by reference.

- Without ref and out (Parameters by Value)

```
static void Main(string[] args)
{
    int a = 10;
    int b = 5;

    WithoutRefOut obj = new WithoutRefOut();

    int x = obj.Update(a, b);

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

Value

```
public class WithoutRefOut
{
    public int Update(int m, int n)
    {
        return m + n;
    }
}
```

Q What is the difference between "out" and "ref" parameters?

OOP

- When to use out and ref?

When you want to return more than one values from a method then you can use out and ref parameters.

- When to use out and when to use ref?

Use out parameter to return a new and fresh value. Use ref to modify an existing value.

- ref keyword is used to pass data in bi-directional way.
- out keyword is used to get data in uni-directional way.

- With ref and out

1. No need to initialize out parameter before passing it.

1. Must initialize ref parameter else error.

2. Out parameter must be initialized before returning.

2. For Ref parameter, Initialization is not necessary before returning.

```
static void Main(string[] args)
{
    int a;

    int b = 5;

    WithRefOut obj = new WithRefOut();

    int x = obj.Update(out a, ref b);

    Console.WriteLine(x);
    Console.ReadLine();
}
public class WithRefOut
{
    public int Update(out int c, ref int d)
    {
        c = 100;
        return c + d;
    }
}
```

Q What is the purpose of "params" keyword?

OOP

- Params keyword is used as a parameter which can take the **VARIABLE** number of parameters.

- It is useful when programmer don't have any prior knowledge about the number of parameters to be used.

You can pass any number of parameters here.

```
static void Main(string[] args)
{
    int sum = Add(5, 10, 15, 20, 30, 40);

    Console.WriteLine(sum);
    Console.ReadLine();
}

public static int Add(params int[] numbers)
{
    int total = 0;

    foreach (int i in numbers)
    {
        total += i;
    }
    return total;
}

//Output: 120
```

Q What is a Constructor? V. IMP.

OOP

- Constructor is a special method of the class that is **AUTOMATICALLY** invoked when an instance of the class is created.
- The name of the constructor must be same as of the class name, otherwise it will be hard to identify it from other methods.

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

```
public class Employee
{
    public Employee()
    {
        Console.WriteLine("Constructor called");
    }
}
static void Main(string[] args)
{

    Employee emp = new Employee();

}

//Output: Constructor called
```

Q What is "this" keyword in C#? When to use it?

OOP

- *this* keyword is used to refer to the CURRENT INSTANCE of the class.

```
class Program
{
    static void Main(string[] args)
    {
        Student std1 = new Student(001, "Jack");
        std1.GetStudent();
    }
}
```

- *this* keyword avoids the name confusion between class fields and constructor parameters.

```
class Student
{
    public int id;
    public string name;
    public Student(int id, string name)
    {
        this.id = id;
        this.name = name;

    }
    public void GetStudent()
    {
        Console.WriteLine(id + " : " + name);
    }
}
```

Q What are Extension Methods in C#? When to use them? V. IMP.

OOP

- Extension method allows you to add new methods in the **existing class without modifying** the source code of the original class.
- USE - Use them when you want to add a method in a class which code you don't have.

```
static void Main(string[] args)
{
    string test = "HelloWorld";

    string left = test.Substring(0, 5);

    Console.WriteLine(left);

    string right = test.RightSubstring(5);

    Console.WriteLine(right);

    //Output: Hello World
}
```

```
public static class StringExtensions
{
    public static string RightSubstring(this String s, int count)
    {
        return s.Substring(s.Length - count, count);
    }
}
```

Q What you mean by Delegate? When to use them? V. IMP.

OOP

Things to remember about delegate:

- A Delegate is a variable that holds the reference to a method or Pointer to a method.
- A delegate can refer to more than one methods of same return type and parameters.
- When to use them?

When we need to pass a method as a parameter.

```
delegate void Calculator(int x, int y);

class Program
{
    public static void Add(int a, int b)
    {
        Console.WriteLine(a + b);
    }
    public static void Mul(int a, int b)
    {
        Console.WriteLine(a * b);
    }

    static void Main(string[] args)
    {
        //Instantiating Delegate
        Calculator calc = new Calculator(Add);

        //Calling method using delegate
        calc(20, 30);
    }
}
```

Q What are Multicast Delegates?

OOP

- A Multicast Delegate in C# is a delegate that holds the references of more than one function.
- A delegate can also be used to execute a series of methods one by one(Chaining of methods).

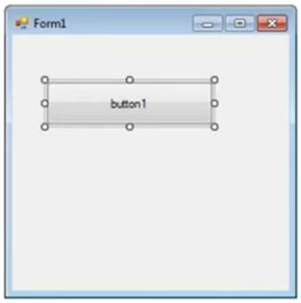
```
delegate void Calculator(int x, int y);

class Program
{
    public static void Add(int a, int b)
    {
        Console.WriteLine(a + b);
    }
    public static void Mul(int a, int b)
    {
        Console.WriteLine(a * b);
    }
    static void Main(string[] args)
    {
        Calculator calc = new Calculator(Add);

        calc += Mul;

        calc(20, 30);
    }
    //Output: 50  600
}
```

Q What are the differences between Events and Delegates?

OOP

- The event is a notification mechanism that depends on delegates

Functions → Delegate → Event



- An event is dependent on a delegate and cannot be created without delegates.
- Event is like a wrapper over the delegate to improve its security.

Q What is the purpose of "using" keyword in C#? **V. IMP.**

OOP

- There are two purpose of using keyword in C#:

1. USING DIRECTIVE

```
using System;
using System.Data.SqlClient;
```

2. USING STATEMENT - The using statement ensures that DISPOSE() method of the class object is called even if an exception occurs.

```
static void Main(string[] args)
{
    using(var connection = new SqlConnection("ConnectionString"))
    {
        var query = "UPDATE YourTable SET Property = Value";
        var command = new SqlCommand(query, connection);

        connection.Open();
        command.ExecuteNonQuery();

        //connection.Dispose();
    }
}
```

Q What is the difference between "is" and "as" operators?

OOP

- The **IS** operator is **USED TO CHECK** the type of an object.

```
static void Main(string[] args)
{
    int i = 5;

    bool check = i is int;

    Console.WriteLine(check);

    //Output: true
}
```

- **AS** operator is used to PERFORM CONVERSION between compatible reference type.

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

    object obj = "Hello";

    string str1 = obj as string;

    Console.WriteLine(str1);

    //Output: Hello
}
```

Q What is the difference between "Readonly" and "Constant" variables (atleast 3)? V. IMP.

OOP

- Constant

```
class Example
{
    public const int myConst = 10;

    public const int myConst1;

    public Example(int b)
    {
        myConst = 20;
    }
}
```

- Readonly

```
class Example
{
    public readonly int myReadonly1 = 100;

    public readonly int myReadonly2;

    public Example(int b)
    {
        myReadonly2 = b * 100;
    }
}
```

1. Using const, we must assign values with declaration, but readonly fields can be assigned in declaration as well as in the constructor part.
2. Constant field value cannot be changed, but Readonly field value can be changed.
3. "const" keyword for Constant and "readonly" keyword is used for Readonly.
4. Constant is a COMPILE time constant, and ReadOnly is a RUNTIME constant.

Q What is "Static" class? When to use it?

OOP

- A static class is a class which object can not be created, and which can not be inherited.

- Use of static class:

Static classes are used as containers for static members like methods, constructors and others.

```
public static class MyCollege
{
    //static fields
    public static string collegeName;
    public static string address;

    //static constructor
    static MyCollege()
    {
        collegeName = "ABC College";
    }

    // static method
    public static void CollegeBranch()
    {
        Console.WriteLine("Computers");
    }
}
```

Q What is the difference between "var" and "dynamic" in C#?

OOP

- VAR - The type of the variable is decided by the compiler at **compile time**.

```
static void Main(string[] args)
{
    var a = 10;

    a = "Interview";

    Console.WriteLine(a);
}
```

- DYNAMIC - The type of the variable is decided at **run time**.

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

    dynamic b = 10;

    b = "Happy";

    Console.WriteLine(b);
}
```