**Object Oriented Programming**

**OOPs Concept:**

We know all that

C# is a **modern fully Object Oriented Programming Language**. So basically OOPs is a **programming paradigm based on classes and objects** to create applications.

**Class:**  Class is a blueprint for creating objects. We can say class is a **collection of properties, member variables and member methods** used to represent logical entities of the program.

* Class is a **user defined data type and also reference type** and acts as a **template for creating objects.**
* Class contains **(variables called 🡪member variables/fields) and (methods called🡪member methods.)**

**Object:** Object is an **instance of Class** and created at runtime. The Object has state and behavior.

State🡪data/property, behavior🡪 functionality

public class Student

{

int roll\_no; //non-static member variable, it is by default private

public string name; //non-static member variable, it is public variable

public static string SchoolName; //static member variable

public int fees { get; set; } //Properties

public void Show\_Name() //Member Methods

{

Console.WriteLine("Student name is : " + name);

}

}

**Features:**

* OOP provides a clear structure for the program.
* OOP provides more readability.
* OOP provides code reusability.
* OOP is faster and easy to execute.
* OOP follows a Bottom-Up programming approach.

**The four fundamentals of OPPs Concept:**

1. **Inheritance**

* **Single Level Inheritance**
* **Hierarchal Level Inheritance**
* **Multi Level Inheritance**
* **Method Hiding**

1. **Abstraction**

* **Abstract Class**
* **Abstract Method**
* **Interface**

1. **Encapsulation**
2. **Polymorphism**

* **Compile Time Polymorphism**
* **Method Overloading**
* **Operator Overloading**
* **Run Time Polymorphism**
* **Method Overriding**

Note: **Simula** (1960) is considered the first object-oriented programming language.

**Smalltalk** is considered the first truly object-oriented programming language.

**Inheritance:**

Inheritance is a process of OOPs in which **one class acquires** all the properties and behaviour from another class.

* Parent Class: The class whose members and properties are inherited called **Base class /Parent Class/Super Class**.
* Child Class: The class which inherits members and properties of another class is called **Derived class/Child Class/Sub Class**.

**Advantages:** We don’t need to recreate methods.

We can reuse methods using inheritance property.

It reduces the complexity of the program.

**Types of Inheritance: Single Level Inheritance, Hierarchal Level Inheritance, Multi Level Inheritance**

* **Single Level Inheritance:** A derived class inherits the properties and behaviour of another single base class.

Single Derived Class: Single Base Class

class BaseClass

{

protected string name = "Hello world"; //protected member variable

protected void Add() //protected member method

{

Console.WriteLine("ADD function");

}

}

Internal class DerivedClass: BaseClass

{

static void Main(string[] args)

{

DerivedClass obj = new DerivedClass();

Console.WriteLine(obj.name);

obj.Add();

Console.ReadLine();

}

}

* **Hierarchal Level Inheritance:** Two or more class inherits the properties and behaviour of another single base class.

class Baseclass

{

public string basename = "Hello Base Class ";

public void Add()

{

Console.WriteLine("Add function");

}

}

class derivedclass1 : Baseclass

{

public string derivedname1 = "Hello derived Class 1";

public void Sub()

{

Console.WriteLine("Sub function");

}

}

class derivedclass2 : Baseclass

{

public string derivedname2 = "Hello derived Class 2";

public void Mul()

{

Console.WriteLine("Mul function");

}

}

internal class Program

{

static void Main(string[] args)

{

derivedclass1 derivedclass1 = new derivedclass1();

derivedclass1.Sub();

Console.WriteLine(derivedclass1.derivedname1);

derivedclass2 derivedclass2 = new derivedclass2();

derivedclass2.Mul();

Console.WriteLine(derivedclass2.derivedname2);

Console.ReadLine();

}

}

* **Multi Level Inheritance:** A class inherits another derived class.

class Greatgrandfather

{

protected void GreatGrandfather1()

{

Console.WriteLine("Grand Father property");

}

}

class Grandfather : Greatgrandfather

{

protected void Grandfather1()

{

Console.WriteLine("Grand Father 2 property");

}

}

class Father : Grandfather

{

protected void Father1()

{

Console.WriteLine("Father property");

}

}

internal class Son : Father

{

static void Main(string[] args)

{

Son son = new Son();

son.GreatGrandfather1();

son.Grandfather1();

son.Father1();

Console.ReadLine();

}

}

* **Method Hiding:**

Method Hiding occurs in inheritance when base class and derived class both have same method and same signature.

When the **object is created of derived class it will hide base class method and will call own its method** and this is called **method hiding in inheritance**. We use ‘new’ keyword in derived class.

**Note**: It we don’t use ‘new; keyword then compiler will show the warning, but program will work fine.

class Baseclass

{

public void Show()

{

Console.WriteLine("This is base class method...");

}

}

class ChildClass : Baseclass

{

public new void Show()

{

Console.WriteLine("This is derived class method...");

}

}

internal class Program

{

static void Main(string[] args)

{

ChildClass childClass = new ChildClass();

childClass.Show();

Console.ReadLine();

}

}

Output: This is derived class method...

If we want to use base class method.

class Baseclass

{

public void Show()

{

Console.WriteLine("This is base class method...");

}

}

class ChildClass : Baseclass

{

public new void Show()

{

base.Show();

}

}

internal class Program

{

static void Main(string[] args)

{

ChildClass childClass = new ChildClass();

childClass.Show();

Console.ReadLine();

}

}

Output: This is base class method...

**Polymorphism:**

Polymorphism is OOPs concept that refers that we can use **same method multiple times but must be different parameters.**

The word Polymorphism is derived from Greek word. Polymorphism is combination of two words **Poly + morph which means Many Forms.**

There are two types of Polymorphism:

* **Compile Time Polymorphism**
* **Run Time Polymorphism**

**Compile Time Polymorphism:** It isachieved by **Method Overloading and Operator Overloading.**

It is also known as **Static Polymorphism/ Static Binding /Early Binding.**

**Run Time Polymorphism:** It isachieved by **Method Overriding.**

It is also known as **Dynamic Polymorphism/ Dynamic Binding /Late Binding.**

* **Method Overloading:**

If there are **two or more methods with same name** but with **different** **parameters** is known as **Method Overloading.**

public void ADD()

{

int a = 10, b = 20, res;

res = a + b;

Console.WriteLine(res);

}

public void ADD(int a, int b)

{

int res = a + b;

Console.WriteLine(res);

}

public void ADD(string a, string b)

{

string res = a + “ ” + b;

Console.WriteLine(res);

}

public void ADD(double a, double b) //work on data type , 2 4 8 16

{

double res = a + b;

Console.WriteLine(res);

}

static void Main(string[] args)

{

Program program = new Program();

program.ADD();

Console.ReadLine();

}

* **Operator Overloading:**

Operator overloading is a ability to use same operator to do various operations. Only **predefined operator** can be overloading.

class NewClass

{

public string str;

public int num;

public static NewClass operator +(NewClass obj1, NewClass obj2)

{

NewClass obj3= new NewClass();

obj3.str = obj1.str + “ ” + obj2.str;

obj3.num = obj1.num+obj2.num;

return obj3;

}

}

internal class Program

{

static void Main(string[] args)

{

NewClass obj1= new NewClass();

obj1.str ="Sudarshan";

obj1.num=20;

NewClass obj2 = new NewClass();

obj2.str ="Sharma";

obj2.num = 30;

NewClass obj3 = new NewClass();

obj3 = obj1 + obj2;

Console.WriteLine(obj3.str);

Console.WriteLine(obj3.num);

Console.ReadLine();

}

}

* **Method Overriding:**

If derived class and base class both have same method, it is called Method overriding.

To perform method overriding, use ‘**virtual’** keyword in base class method and ‘**override’** keyword in derived class.

class Baseclass

{

public virtual void Show()

{

Console.WriteLine("This is base class method...");

}

}

class ChildClass: Baseclass

{

public override void Show()

{

Console.WriteLine("This is derived class method...");

}

}

internal class Program

{

static void Main(string[] args)

{

ChildClass childClass= new ChildClass();

childClass.Show();

Console.ReadLine();

}

}

Output: This is derived class method.

**Abstraction:**

**Abstraction** is a process of **showing only functionality** (**necessary method**) **and hiding the implementation details** to the user.

Means **displaying what is necessary** (**Abstraction**) and **hiding unnecessary details (Encapsulation)** to the outside world.

Hiding can be achieved by using **‘private’** access modifiers**.**

class Calculation

{

private float num1, num2, res; //private member variable

private void Input() //private member method

{

Console.WriteLine("Enter the first num: ");

num1 = Convert.ToInt32(Console.ReadLine());

Console.WriteLine("Enter the first num: ");

num2 = Convert.ToInt32(Console.ReadLine());

}

private void Calculate() //private member method

{

Input();

res = num1 + num2;

Console.WriteLine("Result is : "+ res);

}

public void Show()

{

Calculate();

}

internal class Program

{

static void Main(string[] args)

{

Calculation calculation = new Calculation();

calculation.Show();

Console.ReadLine();

}

}

}

Use override keyword in main class

There are two ways to achieve abstraction

1. Abstract class (0 to 100%)
2. Interface (100%): multi level inheritance

**Abstract Class:**

A class which is declared with the **abstract** keyword is known as an **abstract class**.

We can’t instantiated abstract class and interface.

Abstract class can have **abstract and non-abstract methods**.

Abstract method does not have body.

It needs to be extended (there is need of inheritance)

Abstract class can have modifiers with methods, properties.

public abstract class TataPunch\_Base

{

public abstract void MusicSystem();

public abstract void FlipKey();

public void wheel()

{

Console.WriteLine("normal 4 wheels are in Base Model:");

}

}

public abstract class TataPunch\_Advant

{

public void MusicSystem()

{

Console.WriteLine("Play music");

}

public abstract void FlipKey();

public void wheel()

{

Console.WriteLine("4 wheels are in Base Model:");

}

}

public abstract class TataPunch\_Top

{

public void MusicSystem()

{

Console.WriteLine(" Play music");

}

public void FlipKey()

{

Console.WriteLine("use flipkey");

}

public void wheel()

{

Console.WriteLine("top 4 wheels are in Base Model:");

}

}

internal class Customer: TataPunch\_Base

{

public void Engine()

{

Console.WriteLine("Start Engine...");

}

public override void FlipKey()

{

Console.WriteLine("use flip key");

}

public override void MusicSystem()

{

Console.WriteLine("Use music system");

}

static void Main(string[] args)

{

Customer customer= new Customer();

customer.Engine();

customer.FlipKey();

customer.MusicSystem();

Console.ReadLine();

}

}

Interface: Another way to achieve abstraction is Interface. Interface can contain only properties and methods, not fields.

Interface does not have body.

When a class inherits interface is called **Implements**.

Declare with **interface** keyword.

Interface can’t be instantiate.

interface demo1

{

void Alloywheel(); // by default it is public and abstract method both.

// and this method does not have body, only declaration.

}

interface demo2

{

void Music();

}

internal class Customer:demo1,demo2

{

public void Engine()

{

Console.WriteLine("Start Engine...");

}

static void Main(string[] args)

{

Customer customer= new Customer();

customer.Engine();

Console.ReadLine();

}

public void Music() { }

public void Alloywheel() { }

}

-------------------------------------------

**public** **interface** Drawable

{

**void** draw();

}

**public** **class** Rectangle : Drawable

{

**public** **void** draw()

    {

        Console.WriteLine("drawing rectangle...");

    }

}

**public** **class** Circle : Drawable

{

**public** **void** draw()

    {

        Console.WriteLine("drawing circle...");

    }

}

**public** **class** TestInterface

{

**public** **static** **void** Main()

    {

        Drawable d;

        d = **new** Rectangle();

        d.draw();

        d = **new** Circle();

        d.draw();

    }

}

**Encapsulation:**

Encapsulation is one of the four fundamentals,It is a process **binding the related data**, the **main object of encapsulation is Data Security,** it is a protective shield that **hiding the implementation details** of a class from outside access.

**To achieve the encapsulation:**

Variable must be **private** and these variable can be **accessed only through the method or properties** of their current class. therefore, this is known as **data hiding.**

No Encapsulation and no privacy…

class Person

{

public string Name;

public int Age;

}

internal class Program

{

static void Main(string[] args)

{

Person person = new Person();

person.Name = "Akash";

Console.WriteLine(person.Name);

person.Age = 25;

Console.WriteLine(person.Age);

Console.ReadLine();

}

}

Output:

Akash

25

--------------

class Person

{

private string Name;

private int Age;

public void SetName(string Name) //we use public method to access private fields.

{

if (string.IsNullOrEmpty(Name) == true)

{

Console.WriteLine("Name is required !! ");

}

else

{

this.Name = Name;

}

}

public void GetName()

{

if (string.IsNullOrEmpty(Name) == true)

{

}

else

{

Console.WriteLine("Your name is : " + this.Name); ;

}

}

public void SetAge(int Age) //we use public method to access private fields.

{

if (Age > 0)

{

this.Age = Age;

}

else

{

Console.WriteLine("Age should not be negative or zero !!");

}

}

public void GetAge()

{

if (Age > 0)

{

Console.WriteLine("Your age is " + this.Age);

}

else

{

}

}

internal class Program

{

static void Main(string[] args)

{

Person person = new Person();

person.SetName("Karan");

person.GetName();

person.SetAge(25);

person.GetAge();

Console.ReadLine();

}

}

}

Output:

Karan

25

* OOP provides of code reusability.
* OOP is faster and easy to execute.
* OOP follows Bottom-Up programming approach.

public class Person

{

// Member Variables (State)

public string Name;

public int Age;

// Member Methods (Behavior)

public void DisplayInfo()

{

Console.WriteLine($"Name: {Name}, Age: {Age}");

}

}

**The four fundamentals of OPPs Concept:**

1. **Inheritance**

* **Single Level Inheritance**
* **Hierarchal Level Inheritance**
* **Multi Level Inheritance**
* **Method Hiding**

1. **Abstraction**

* **Abstract Class**
* **Abstract Method**
* **Interface**

1. **Encapsulation**
2. **Polymorphism**

* **Compile Time Polymorphism**
* **Method Overloading**
* **Operator Overloading**
* **Run Time Polymorphism**
* **Method Overriding**

Note: **Simula** (1960) is considered the first object-oriented programming language.

**Smalltalk** is considered the first truly object-oriented programming language.

**Inheritance:**

Inheritance is a process of OOPs in which **one class acquires** all the properties and behaviour from another class.

* Parent Class: The class whose members and properties are inherited called **Base class /Parent Class/Super Class**.
* Child Class: The class which inherits members and properties of another class is called **Derived class/Child Class/Sub Class**.

**Advantages:** We don’t need to recreate methods.

We can reuse methods using inheritance property.

It reduces the complexity of program.

**Types of Inheritance: Single Level Inheritance, Hierarchal Level Inheritance, Multi Level Inheritance**

* **Single Level Inheritance:** A derived class inherits the properties and, behaviour of another single base class.

Single Derived Class: Single Base Class

class BaseClass

{

protected string name = "Hello world"; //protected member variable

protected void Add() //protected member method

{

Console.WriteLine("ADD fucntion");

}

}

Internal class DerivedClass: BaseClass

{

static void Main(string[] args)

{

DerivedClass obj = new DerivedClass();

Console.WriteLine(obj.name);

obj.Add();

Console.ReadLine();

}

}

* **Hierarchal Level Inheritance:** Two or more class inherits the properties and behaviour of another single base class.

class Baseclass

{

public string basename = "Hello Base Class ";

public void Add()

{

Console.WriteLine("Add function");

}

}

class derivedclass1 : Baseclass

{

public string derivedname1 = "Hello derived Class 1";

public void Sub()

{

Console.WriteLine("Sub function");

}

}

class derivedclass2 : Baseclass

{

public string derivedname2 = "Hello derived Class 2";

public void Mul()

{

Console.WriteLine("Mul function");

}

}

internal class Program

{

static void Main(string[] args)

{

derivedclass1 derivedclass1 = new derivedclass1();

derivedclass1.Sub();

Console.WriteLine(derivedclass1.derivedname1);

derivedclass2 derivedclass2 = new derivedclass2();

derivedclass2.Mul();

Console.WriteLine(derivedclass2.derivedname2);

Console.ReadLine();

}

}

* **Multi Level Inheritance:** A class inherits another derived class.

class Greatgrandfather

{

protected void GreatGrandfather1()

{

Console.WriteLine("Grand Father property");

}

}

class Grandfather : Greatgrandfather

{

protected void Grandfather1()

{

Console.WriteLine("Grand Father 2 property");

}

}

class Father : Grandfather

{

protected void Father1()

{

Console.WriteLine("Father property");

}

}

internal class Son : Father

{

static void Main(string[] args)

{

Son son = new Son();

son.GreatGrandfather1();

son.Grandfather1();

son.Father1();

Console.ReadLine();

}

}

* **Method Hiding:**

Method Hiding occurs in inheritance when base class and derived class both have same method and same signature.

When the **object is created of derived class it will hide base class method and will call own its method** and this is called **method hiding in inheritance**. We use ‘new’ keyword in derived class.

**Note**: It we don’t use ‘new; keyword then compiler will show the warning, but program will work fine.

class Baseclass

{

public void Show()

{

Console.WriteLine("This is base class method...");

}

}

class ChildClass : Baseclass

{

public new void Show()

{

Console.WriteLine("This is derived class method...");

}

}

internal class Program

{

static void Main(string[] args)

{

ChildClass childClass = new ChildClass();

childClass.Show();

Console.ReadLine();

}

}

Output: This is derived class method...

If we want to use base class method.

class Baseclass

{

public void Show()

{

Console.WriteLine("This is base class method...");

}

}

class ChildClass : Baseclass

{

public new void Show()

{

base.Show();

}

}

internal class Program

{

static void Main(string[] args)

{

ChildClass childClass = new ChildClass();

childClass.Show();

Console.ReadLine();

}

}

Output: This is base class method...

**Polymorphism:**

Polymorphism is OOPs concept that refers that we can use **same method multiple times but must be different parameters.**

The word Polymorphism is derived from Greek word. Polymorphism is combination of two words **Poly + morph which means Many Forms.**

There are two types of Polymorphism:

* **Compile Time Polymorphism**
* **Run Time Polymorphism**

**Compile Time Polymorphism:** It isachieved by **Method Overloading and Operator Overloading.**

It is also known as **Static Polymorphism/ Static Binding /Early Binding.**

**Run Time Polymorphism:** It isachieved by **Method Overriding.**

It is also known as **Dynamic Polymorphism/ Dynamic Binding /Late Binding.**

* **Method Overloading:**

If there are **two or more methods with same name** but with **different** **parameters** is known as **Method Overloading.**

public void ADD()

{

int a = 10, b = 20, res;

res = a + b;

Console.WriteLine(res);

}

public void ADD(int a, int b)

{

int res = a + b;

Console.WriteLine(res);

}

public void ADD(string a, string b)

{

string res = a + “ ” + b;

Console.WriteLine(res);

}

public void ADD(double a, double b) //work on data type , 2 4 8 16

{

double res = a + b;

Console.WriteLine(res);

}

static void Main(string[] args)

{

Program program = new Program();

program.ADD();

Console.ReadLine();

}

* **Operator Overloading:**

Operator overloading is a ability to use same operator to do various operations. Only **predefined operator** can be overloading.

class NewClass

{

public string str;

public int num;

public static NewClass operator +(NewClass obj1, NewClass obj2)

{

NewClass obj3= new NewClass();

obj3.str = obj1.str + “ ” + obj2.str;

obj3.num = obj1.num+obj2.num;

return obj3;

}

}

internal class Program

{

static void Main(string[] args)

{

NewClass obj1= new NewClass();

obj1.str ="Sudarshan";

obj1.num=20;

NewClass obj2 = new NewClass();

obj2.str ="Sharma";

obj2.num = 30;

NewClass obj3 = new NewClass();

obj3 = obj1 + obj2;

Console.WriteLine(obj3.str);

Console.WriteLine(obj3.num);

Console.ReadLine();

}

}

* **Method Overriding:**

If derived class and base class both have same method, it is called Method overriding.

To perform method overriding, use ‘**virtual’** keyword in base class method and ‘**override’** keyword in derived class.

class Baseclass

{

public virtual void Show()

{

Console.WriteLine("This is base class method...");

}

}

class ChildClass: Baseclass

{

public override void Show()

{

Console.WriteLine("This is derived class method...");

}

}

internal class Program

{

static void Main(string[] args)

{

ChildClass childClass= new ChildClass();

childClass.Show();

Console.ReadLine();

}

}

Output: This is derived class method.

**Abstraction:**

**Abstraction** is a process of **showing only functionality** (**necessary method**) **and hiding the implementation details** to the user.

Means **displaying what is necessary** (**Abstraction**) and **hiding unnecessary details (Encapsulation)** to the outside world.

Hiding can be achieved by using **‘private’** access modifiers**.**

class Calculation

{

private float num1, num2, res; //private member variable

private void Input() //private member method

{

Console.WriteLine("Enter the first num: ");

num1 = Convert.ToInt32(Console.ReadLine());

Console.WriteLine("Enter the first num: ");

num2 = Convert.ToInt32(Console.ReadLine());

}

private void Calculate() //private member method

{

Input();

res = num1 + num2;

Console.WriteLine("Result is : "+ res);

}

public void Show()

{

Calculate();

}

internal class Program

{

static void Main(string[] args)

{

Calculation calculation = new Calculation();

calculation.Show();

Console.ReadLine();

}

}

}

Use override keyword in main class

There are two ways to achieve abstraction

1. Abstract class (0 to 100%)
2. Interface (100%): multi level inheritance

**Abstract Class:**

A class which is declared with the **abstract** keyword is known as an **abstract class**.

We can’t instantiated abstract class and interface.

Abstract class can have **abstract and non-abstract methods**.

Abstract method does not have body.

It needs to be extended (there is need of inheritance)

Abstract class can have modifiers with methods, properties.

public abstract class TataPunch\_Base

{

public abstract void MusicSystem();

public abstract void FlipKey();

public void wheel()

{

Console.WriteLine("normal 4 wheels are in Base Model:");

}

}

public abstract class TataPunch\_Advant

{

public void MusicSystem()

{

Console.WriteLine("Play music");

}

public abstract void FlipKey();

public void wheel()

{

Console.WriteLine("4 wheels are in Base Model:");

}

}

public abstract class TataPunch\_Top

{

public void MusicSystem()

{

Console.WriteLine(" Play music");

}

public void FlipKey()

{

Console.WriteLine("use flipkey");

}

public void wheel()

{

Console.WriteLine("top 4 wheels are in Base Model:");

}

}

internal class Customer: TataPunch\_Base

{

public void Engine()

{

Console.WriteLine("Start Engine...");

}

public override void FlipKey()

{

Console.WriteLine("use flip key");

}

public override void MusicSystem()

{

Console.WriteLine("Use music system");

}

static void Main(string[] args)

{

Customer customer= new Customer();

customer.Engine();

customer.FlipKey();

customer.MusicSystem();

Console.ReadLine();

}

}

Interface: Another way to achieve abstraction is Interface. Interface can contain only properties and method, not fields.

Interface does not have body.

When a class inherits interface is called **Implements**.

Declare with **interface** keyword.

Interface can’t be instantiate.

interface demo1

{

void Alloywheel(); // by default it is public and abstract method both.

// and this method does not have body, only declaration.

}

interface demo2

{

void Music();

}

internal class Customer:demo1,demo2

{

public void Engine()

{

Console.WriteLine("Start Engine...");

}

static void Main(string[] args)

{

Customer customer= new Customer();

customer.Engine();

Console.ReadLine();

}

public void Music() { }

public void Alloywheel() { }

}

-------------------------------------------

**public** **interface** Drawable

{

**void** draw();

}

**public** **class** Rectangle : Drawable

{

**public** **void** draw()

    {

        Console.WriteLine("drawing rectangle...");

    }

}

**public** **class** Circle : Drawable

{

**public** **void** draw()

    {

        Console.WriteLine("drawing circle...");

    }

}

**public** **class** TestInterface

{

**public** **static** **void** Main()

    {

        Drawable d;

        d = **new** Rectangle();

        d.draw();

        d = **new** Circle();

        d.draw();

    }

}

**Encapsulation:**

Encapsulation is one of the four fundamentals,It is a process **binding the related data**, the **main object of encapsulation is Data Security,** it is a protective shield that **hiding the implementation details** of a class from outside access.

**To achieve the encapsulation:**

Variable must be **private** and these variable can be **accessed only through the method or properties** of their current class. therefore, this is known as **data hiding.**

No Encapsulation and no privacy…

class Person

{

public string Name;

public int Age;

}

internal class Program

{

static void Main(string[] args)

{

Person person = new Person();

person.Name = "Akash";

Console.WriteLine(person.Name);

person.Age = 25;

Console.WriteLine(person.Age);

Console.ReadLine();

}

}

Output:

Akash

25

--------------

class Person

{

private string Name;

private int Age;

public void SetName(string Name) //we use public method to access private fields.

{

if (string.IsNullOrEmpty(Name) == true)

{

Console.WriteLine("Name is required !! ");

}

else

{

this.Name = Name;

}

}

public void GetName()

{

if (string.IsNullOrEmpty(Name) == true)

{

}

else

{

Console.WriteLine("Your name is : " + this.Name); ;

}

}

public void SetAge(int Age) //we use public method to access private fields.

{

if (Age > 0)

{

this.Age = Age;

}

else

{

Console.WriteLine("Age should not be negative or zero !!");

}

}

public void GetAge()

{

if (Age > 0)

{

Console.WriteLine("Your age is " + this.Age);

}

else

{

}

}

internal class Program

{

static void Main(string[] args)

{

Person person = new Person();

person.SetName("Karan");

person.GetName();

person.SetAge(25);

person.GetAge();

Console.ReadLine();

}

}

}

Output:

Karan

25