Inheritance in C#

What is Inheritance in C# ?

- In C#, inheritance allows us to create a new class from an existing class.
- The class from which a new class is created is known as the base class (parent or superclass). And, the new class is called derived class (child or subclass).
- The derived class inherits the fields (data) and methods of the base class.
- This helps with the code reusability in C#.

How to perform inheritance in C#?

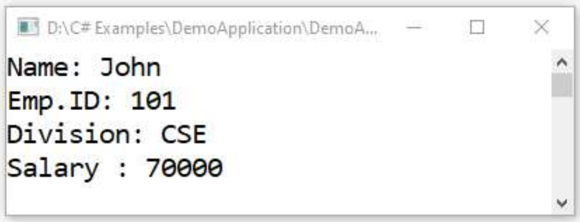
In C#, we use the : symbol to perform inheritance. For example,

```
using System;
namespace DemoApplication
{
    class Employee
    {
        //fields
        //methods
    }
    class Manager : Employee
    {
        // inherit fields and methods of Employee class
        // fields and methods of Manager also
    }
    class Program
    {
        static void Main(string[] args)
        {

        }
    }
}
```

Here, we are inheriting the derived class Manager from the base class Employee. The Manager class can now access the fields and methods of Employee class.

Example 2: Employee and Manager Inheritance Program with fields and methods.

```
using System;
namespace DemoApplication
{
    class Employee
    {
        public string name;
        public int empid;
        public double basesalary;

        public void saveEmpDetails(string name, int empid,double basesalary)
        {
            this.name = name;
            this.empid = empid;
            this.basesalary = basesalary;
        }
    }
    class Manager : Employee
    {
        public string division;
        public double incentive;

        public void saveManagerDetails(string division,double incentive)
        {
            this.division = division;
            this.incentive = incentive;
        }

        public void viewDetails()
        {
            Console.WriteLine("Name: " + name);
            Console.WriteLine("Emp.ID: " + empid);
            Console.WriteLine("Division: " + division);
            Console.WriteLine("Salary : " + (basesalary + incentive));
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Manager manager = new Manager();
            manager.saveEmpDetails("John", 101, 40000);
            manager.saveManagerDetails("CSE", 30000);
            manager.viewDetails();
            Console.ReadKey();
        }
    }
}
```



```
Name: John
Emp.ID: 101
Division: CSE
Salary : 70000
```

Here, we are inheriting the derived class Manager from the base class Employee. The Manager class can now access the fields and methods such as name, empid, basicsalary and saveEmpDetails() of Employee class.

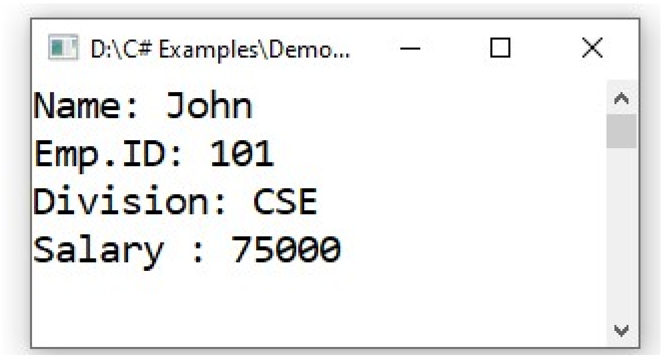
In Main() method, we are creating object for the derived class called Manager and we can access the base class fields and methods using the Manager object. The is the benefit of applying inheritance in C# application development (Code Reusability).

Constructors in Inheritance

The base class constructor cannot be inherited to the derived class, instead it can be accessed via

: base() method call from derived class constructors.

```
using System;
namespace DemoApplication
{
    class Employee
    {
        public string name;
        public int empid;
        public double basesalary;

        public Employee(string name, int empid,double basesalary)
        {
            this.name = name;
            this.empid = empid;
            this.basesalary = basesalary;
        }
    }
    class Manager : Employee
    {
        public string division;
        public double incentive;

        public Manager(string name, int empid, double basesalary,string
                   division,double incentive) : base(name, empid, basesalary)
        {
            this.division = division;
            this.incentive = incentive;
        }

        public void viewDetails()
        {
            Console.WriteLine("Name:  " + name);
            Console.WriteLine("Emp.ID:  " + empid);
            Console.WriteLine("Division:  " + division);
            Console.WriteLine("Salary : " + (basesalary + incentive));
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Manager manager = new Manager("John",101,50000,"CSE",25000);
            manager.viewDetails();
            Console.ReadKey();
        }
    }
}
```



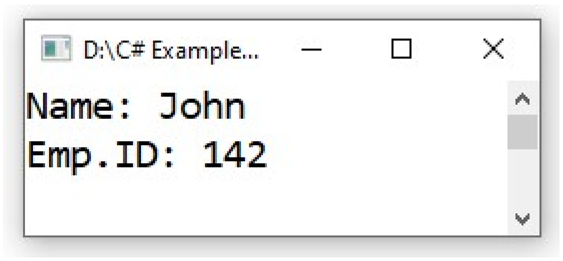
```
Name: John
Emp.ID: 101
Division: CSE
Salary : 75000
```

Here, `: base(name, empid, basesalary)` method call in derived class Manager constructor call the base class constructor and the arguments name, empid, basesalary values will be pass on to the parameters of the base class Employee constructors to initialize the same.

The base keyword in Inheritance

In C#, base keyword is used to access fields, constructors and methods of base class.

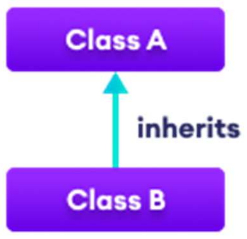
```
using System;
namespace DemoApplication
{
    class Employee
    {
        public string name;
        public int empid;

        public Employee(string name, int empid)
        {
            this.name = name;
            this.empid = empid;
        }
    }
    class Manager : Employee
    {
        public Manager(string name, int empid) : base(name,empid)
        {

        }
        public void view()
        {
            Console.WriteLine("Name:  " +  base.name);
            Console.WriteLine("Emp.ID:  " + base.empid);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Manager manager = new Manager("John",142);
            manager.view();
            Console.ReadKey();
        }
    }
}
```



```
Name:  John
Emp.ID:  142
```

Types of Inheritance

1. **Single Inheritance**
   - In single inheritance, a single derived class inherits from a single base class.



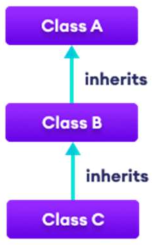
Example:

```
using System;
namespace DemoApplication
{
    class Employee
    {
        public string name;
        public int empid;
        public Employee(string name, int empid)
```

```
                {
                    this.name = name;
                    this.empid = empid;
                }
            }
            class Manager : Employee
            {
                public Manager(string name, int empid) : base(name,empid)
                {
                }
                public void view()
                {
                    Console.WriteLine("Name:  " +  name);
                    Console.WriteLine("Emp.ID:  " + empid);
                }
            }
            class Program
            {
                static void Main(string[] args)
                {
                    Manager manager = new Manager("John",142);
                    manager.view();
                    Console.ReadKey();
                }
            }
        }
```

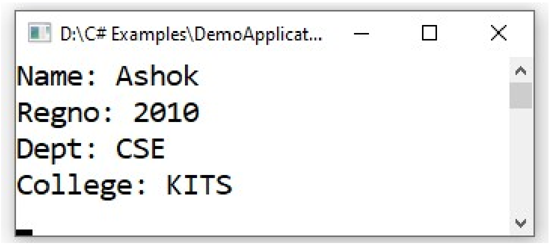
2. Multilevel Inheritance

- In multilevel inheritance, a derived class inherits from a base and then the same derived class acts as a base class for another class.



Example Program:

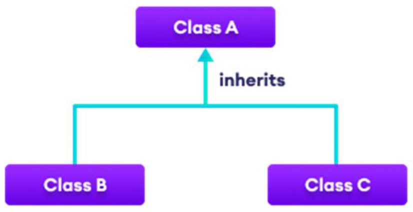
```
using System;
namespace DemoApplication
{
    class College
    {
        protected string collegename;
        public College(string collegename)
        {
            this.collegename = collegename;
        }
    }
    class Department : College
    {
        protected string deptname;
        public Department(string deptname, string collegename) : base(collegename)
        {
            this.deptname = deptname;
        }
    }
    class Student : Department
    {
        string name;
        int regno;
        public Student(string name, int regno, string deptname, string collegename)
                                                    : base(deptname, collegename)
        {
            this.name = name;
            this.regno = regno;
        }
        public void view()
        {
            Console.WriteLine("Name:  " + name);
            Console.WriteLine("Regno:  " + regno);
            Console.WriteLine("Dept:  " + deptname);
            Console.WriteLine("College:  " + collegename);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Student student = new Student("Ashok", 2010, "CSE", "KITS");
            student.view();
            Console.ReadKey();
        }
    }
}
```



```
D:\C# Examples\DemoApplicat...
Name:  Ashok
Regno:  2010
Dept:  CSE
College:  KITS
```

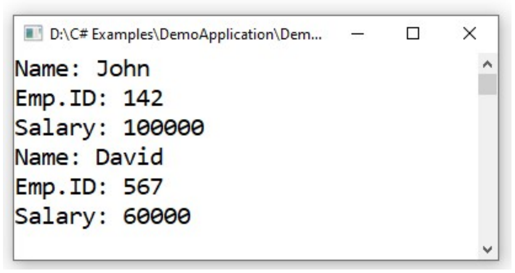
3. Hierarchical Inheritance

- In hierarchical inheritance, multiple derived classes inherit from a single base class.



Example:

```
using System;
namespace DemoApplication
{
    class Employee
    {
        public string name;
        public int empid;
        public Employee(string name, int empid)
        {
            this.name = name;
            this.empid = empid;
        }
    }
    class Manager : Employee
    {
        double salary;
        public Manager(string name, int empid, double salary) : base(name, empid)
        {
            this.salary = salary;
        }
        public void view()
        {
            Console.WriteLine("Name:  " + name);
            Console.WriteLine("Emp.ID:  " + empid);
            Console.WriteLine("Salary:  " + salary);
        }
    }
    class Engineer : Employee
    {
        double salary;
        public Engineer(string name, int empid, double salary) : base(name, empid)
        {
            this.salary = salary;
        }
        public void view()
        {
            Console.WriteLine("Name:  " + name);
            Console.WriteLine("Emp.ID:  " + empid);
            Console.WriteLine("Salary:  " + salary);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Manager manager = new Manager("John", 142,100000);
            manager.view();
            Engineer engineer = new Engineer("David", 567, 60000);
            engineer.view();
            Console.ReadKey();
        }
    }
}
```


D:\C# Examples\DemoApplication\Dem...

```
Name: John
Emp.ID: 142
Salary: 100000
Name: David
Emp.ID: 567
Salary: 60000
```

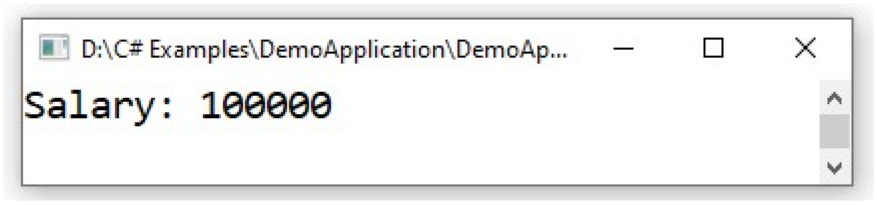
Method Overriding in C# Inheritance

If the same method is present in both the base class and the derived class, the method in the derived class overrides the method in the base class. This is called method overriding in C#.

- The Method Overriding in C# can be achieved using override & virtual keywords and the inheritance principle.
- If we want to change the behavior of the base class method in a derived class, then we need to use method overriding.
- The base class method we want to override in the derived class needs to be defined with a virtual keyword.
- We need to use the override keyword in the derived class while defining the method with the same name and parameters then only we can override the base class method in a derived class.

For example,

```
using System;
namespace DemoApplication {

     class Employee
      {
        protected int empid;
        protected double salary;
        public Employee(int empid, double salary) {
            this.empid = empid;
            this.salary = salary;
        }
        public virtual void calcSalary() {
            Console.WriteLine("Salary: " + salary);
        }
    }
    class Manager : Employee
      {
        double incentive;
        public Manager(int empid, double salary, double incentive) :
                                                    base(empid, salary)
        {
            this.incentive = incentive;
        }
        public override void calcSalary() {
            Console.WriteLine("Salary: " + (salary + incentive));
        }
    }
    class Program {
        static void Main(string[] args) {
            Manager manager = new Manager(142, 75000, 25000);
            manager.calcSalary();
            Console.ReadKey();
        }
    }
}
```



```
Salary: 100000
```

If you observe the above code snippet, we created two classes (“**Employee**”, and “**Manager**”), and the derived class **(Manager)** is inheriting the properties from the base class (**Employee**). We are overriding the base class method **calcSalary()** in the derived class by creating a method with the same name and parameters, and this is called a **method overriding** in c#.

Here, we defined a **calcSalary()** method with a **virtual** keyword in the base class to allow the derived class to override that method using the **override** keyword.

As discussed, only the methods with a **virtual** keyword in the base class are allowed to override in the derived class using the **override** keyword.

Polymorphism

In c#, Polymorphism means providing an ability to take more than one form, and it’s one of the main pillar concepts of object-oriented programming after encapsulation and inheritance.

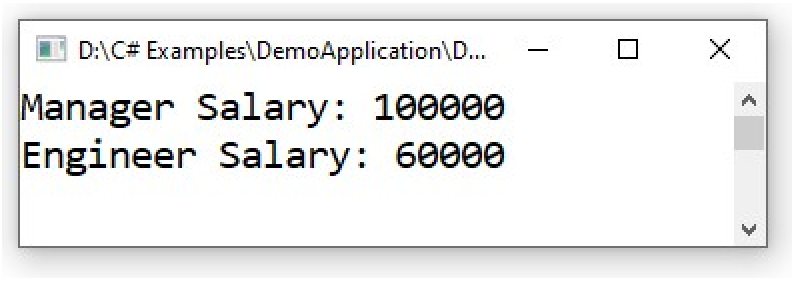
In c#, Run Time Polymorphism means overriding a base class method in the derived class by creating a similar function. This can be achieved by using override & virtual keywords and the base class reference variable.

Here, the base class reference variable take different actions based on the object that is assigned to that, with this base class reference variable if we call any overridden methods then it is called as Run Time Polymorphism.

Note: The base class reference variable can also be called as polymorphic variable.

For Example,

```
using System;
namespace DemoApplication
{
    class Employee
    {
        protected int empid;
        protected double salary;
        public Employee(int empid, double salary) {
            this.empid = empid;
            this.salary = salary;
        }
        public virtual void calcSalary() {
            Console.WriteLine("Basic Salary: " + salary);
        }
    }
    class Manager : Employee {
        double incentive;
        public Manager(int empid, double salary, double incentive):
                                                    base(empid, salary)
        {
            this.incentive = incentive;
        }
        public override void calcSalary() {
            Console.WriteLine("Manager Salary: " + (salary + incentive));
        }
    }
    class Engineer: Employee
    {
        double bonus;
        public Engineer(int empid, double salary, double bonus) : base(empid,
                                                                salary)
        {
            this.bonus = bonus;
        }
        public override void calcSalary()
        {
            Console.WriteLine("Engineer Salary: " + (salary + bonus));
        }
    }
    class Program
    {
        static void Main(string[] args) {
            //polymorphic variable
            Employee emp = null;

            emp = new Manager(142, 75000, 25000);
            emp.calcSalary();

            emp = new Engineer(2001, 45000, 15000);
            emp.calcSalary();

            Console.ReadKey();
        }
    }
}
```



```
Manager Salary: 100000
Engineer Salary: 60000
```

If you observe the above code snippet, we created two classes ("**Employee**", and "**Manager**"), and the derived class **(Manager)** is inheriting the properties from the base class (**Employee**). We are overriding the base class method **calcSalary()** in the derived class by creating a method with the same name and parameters, and this is called a **method overriding** in c#.

Here, we defined a **calcSalary()** method with a **virtual** keyword in the base class to allow the derived class to override that method using the **override** keyword.

Moreover, we used base class reference variable (`Employee emp = null;`) to perform the Run Time Polymorphism, When emp is assigned with Manager object, it calls the Manager's calcSalary() method and when emp is assigned with Engineer object, it calls the Engineer's calcSalary() method. Hence, emp takes different actions depending on the type of object it is referring to since it is a polymorphic reference variable. This run time switching is called Run Time Polymorphism.

The Object Class

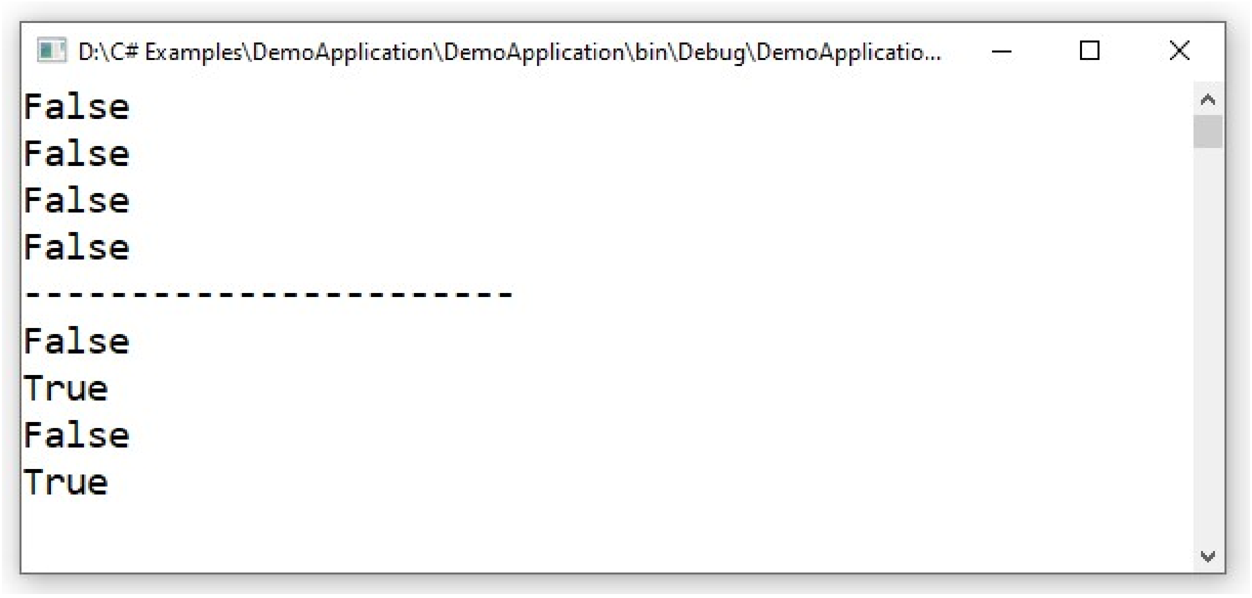
The Object class is base class of all the classes in C# and .NET Framework. A variable of type Object class can be used to store any objects in C#.

The Object class has five methods:

- GetType()
    - Returns the unique type of a data.
- Equals()
    - Tests for data equality (if it is overridden) otherwise it can return True for different instances of the same object, and this is the most commonly overridden method.
- ReferenceEquals()
    - Tests whether or not two objects are the same instance and cannot be overridden.
- ToString
    - To convert any data as String format
- GetHashCode
    - To return unique hash code for an object

Example:

```
using System;
namespace DemoApplication
{
    class Student
    {
        int regno;
        string name;
        public Student(string name,int regno)
        {
            this.name = name;
            this.regno = regno;
        }
    }
    class Program
    {
        static void Main(string[] args) {
            Student s1 = new Student("John", 101);
            Student s2 = new Student("John", 101);
            Student s3 = new Student("David", 102);
            Console.WriteLine(ReferenceEquals(s1, s2));
            Console.WriteLine(ReferenceEquals(s1, s3));
            Console.WriteLine(s1.Equals(s2));
            Console.WriteLine(s1.Equals(s3));
            Console.WriteLine("-----------------------");
            s1 = s3;
            Console.WriteLine(ReferenceEquals(s1, s2));
            Console.WriteLine(ReferenceEquals(s1, s3));
            Console.WriteLine(s1.Equals(s2));
            Console.WriteLine(s1.Equals(s3));
            Console.ReadKey();
        }
    }
}
```


D:\C# Examples\DemoApplication\DemoApplication\bin\Debug\DemoApplicatio...

```
False
False
False
False
-----------------------
False
True
False
True
```

In the above example program, the both Equals() and ReferenceEquals() used for comparing object instances not the data inside objects, hence it returns “False”.

However, after assigning s1 = s3; both methods return “True”, because the both references s1 and s3 is now pointing to same object. Hence, by default both methods compare the reference (address) only not the contents of the object.

Overriding Equals()

To compare the data content of two objects, we must override the Equals() method and write our own logic for the comparison. This is not possible with ReferenceEquals() method.

For Example:

```
using System;
namespace DemoApplication
{
    class Student
    {
        int regno;
        string name;
        public Student(string name,int regno)
        {
            this.name = name;
            this.regno = regno;
        }
        public bool Equals(Student other)
        {
            return this.regno == other.regno && this.name == other.name;
        }
    }
    class Program
    {
        static void Main(string[] args) {
            Student s1 = new Student("John", 101);
            Student s2 = new Student("John", 101);
            Console.WriteLine(s1.Equals(s2));
            Console.ReadKey();
        }
    }
}
```

In the above example program, the Equals() is overridden as given below and we have included the logic for comparision with respect to name and regno of the students. Hence, it returns "True" for the method call `s1.Equals(s2)`

```
        public bool Equals(Student other)
        {
            return this.regno == other.regno && this.name == other.name;
        }
```

Compare two String objects:

The Equals() method can be used to compare two String object contents, since this method is overridden in String class of C#.

Note: Basically, there is no difference between string and System.String in C#. "string" is just an alias of System.String class.

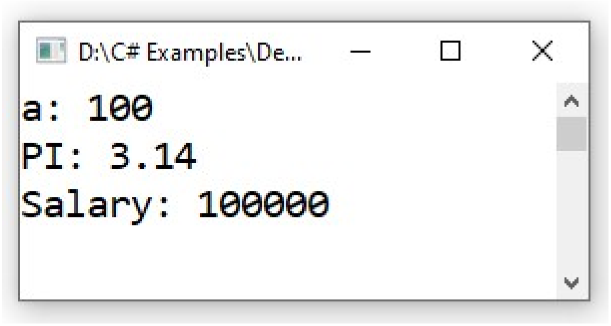
For Example:

```
  class Program
   {
       static void Main(string[] args) {
           string st1 = "JAVA";
           string st2 = "JAVA";
           Console.WriteLine(st1.Equals(st2));

           Console.ReadKey();
       }
   }
```

ToString() method

This method used to convert any data in a String format.

```
class Program
    {
        static void Main(string[] args) {
            int a = 100;
            float pi = 3.14f;
            double salary = 100000;

            string s1 = a.ToString();
            string s2 = pi.ToString();
            string s3 = salary.ToString();

            Console.WriteLine("a: " + s1);
            Console.WriteLine("PI: " + s2);
            Console.WriteLine("Salary: " + s3);
            Console.ReadKey();
        }
    }
```



```
a: 100
PI: 3.14
Salary: 100000
```

Access Modifiers

In C#, there are 4 basic types of access modifiers.

Access Modifier	Description
public	The code is accessible for all classes.
private	The code is only accessible within the same class

Access Modifier	Description
protected	The code is accessible within the same class, or in a class that is inherited from that class.
internal	The code is only accessible within its own assembly(namespace), but not from another assembly.
protected internal	It is used to specify that access is limited to the current assembly or types derived from the containing class.
private protected	It is used to specify that access is limited to the containing class or types derived from the containing class within the current assembly.

private Modifier

If you declare a field with a private access modifier, it can only be accessed within the same class:

```
using System;
namespace DemoApplication {
    class Car {
        private string brand;
        private double price;
    }
    class Program{
        static void Main(string[] args) {
            Car obj = new Car();
            obj.brand = "Creta";        //Error due to private
            obj.price = 1500000;        //Error due to private
            Console.ReadKey();
        }
    }
}
```

public Modifier

If you declare a field with a public access modifier, it is accessible for all classes:

```
using System;
namespace DemoApplication {
    class Car {
        public string brand;
        public double price;
    }
    class Program{
        static void Main(string[] args) {
            Car obj = new Car();
            obj.brand = "Creta";      //Accessible due to public
            obj.price = 1500000;      //Accessible due to public
            Console.ReadKey();
        }
    }
}
```

Refer: **https://www.tutlane.com/tutorial/csharp/csharp-access-modifiers-public-private-protected-internal**