Static Keyword in C# with Examples

Why do we need Static Keyword in C#?

If you ask this question to any developers, they most probably answer you that the static keyword is used in Factory Design Pattern, Singleton Design Pattern as well as used for data sharing, etc. But I think, the static keyword is used for three basic purposes. And in this article, we will discuss these three purposes in detail. I hope you are going to enjoy this article.

Example to understand the Static Keyword in C#:

Let us understand the need and use of the C# Static Keyword with an example. First, create a console application with the name StaticKeyowrdDemo.

CountryMaster.cs:

Once you created the Console application, then create a class file with the name **CountryMaster.cs** and then copy and paste the following code into it. Here we have created the CountryMaster class with three properties and one method. The CountryCode property is going to hold the three-letter symbols of the country like IND while the CountryName property holds the full country name like India. The ComputerName property has the logic to retrieve the current machine name. The Insert Method inserts the country record into the database and while inserting it also uses the ComputerName property to tell that from which computer this record is being inserted.

```
namespace StaticKeyowrdDemo{
    public class CountryMaster {
        public string CountryCode { get; set; }
        public string CountryName { get; set; }
        private string ComputerName {
            get{
                return System.Environment.MachineName;
            }
        }
        public void Insert() {
            //Logic to Insert the Country Details into the Database
            //ComputerName property tells from which computer the Record is being Inserted
        }
    }
}
```

Customer.cs

Now, create a new class file with the name **Customer.cs** and then copy and paste the following code into it.

```
namespace StaticKeyowrdDemo{
    public class Customer{
        public string CustomerCode { get; set; }
        public string CustomerName { get; set; }
        private string MachineName = "";
        private bool IsEmpty(string value){
            if(value.Length > 0){
                return true;
            }
            return false;
        }
        public void Insert() {
            if(IsEmpty(CustomerCode) && IsEmpty(CustomerName)){
                //Insert the data
            }
        }
    }
}
```

Explanation of Above Code:

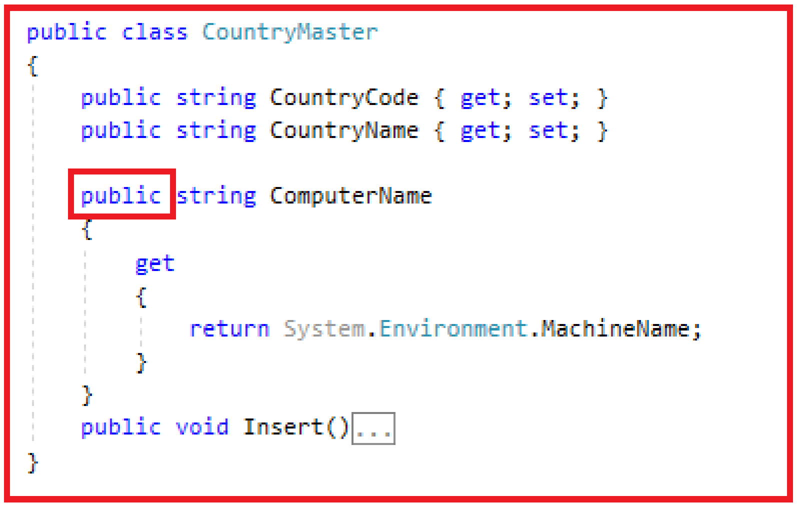
In the above code, the CustomerCode property is going to hold the three-letter code of the customer, for example, AB1 while the CustomerName property holds the customer's name for example Pranaya. The IsEmpty method accepts one value and then checks if the value is empty or not. If not empty then return true else return false. The Insert method simply checks

if both CustomerCode and CustomerName are not empty then insert the customer record into the database.

Here, the problem is with the MachineName variable. The MachineName should have the current computer name while inserting the customer data into the database so that we can track from which machine this customer data was inserted.

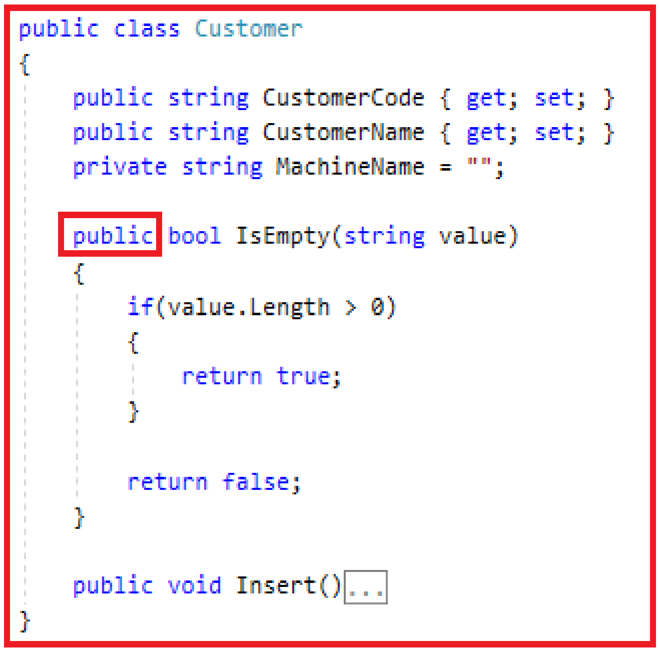
If you remember, the CountryMaster class has the logic to retrieve the computer name. Rather than writing the duplicate logic here, we should go and use the logic which is already written in the CountryMaster class, so that we can avoid rewriting the same code.

If you check the ComputerName property in the class CountryMaster class, then you will see that it is private. So, in order to use that property inside the Customer class, first of all, we need to change that property to the public as shown in the below image.



```
public class CountryMaster
{
    public string CountryCode { get; set; }
    public string CountryName { get; set; }

    public string ComputerName
    {
        get
        {
            return System.Environment.MachineName;
        }
    }
    public void Insert()...
}
```

Again, while inserting the CountryMaster record into the database, we also need to check both CountryCode and CountryName properties should not be empty. To check if empty or not, we would also like to use the IsEmpty method which is defined inside the Customer class rather than writing the same logic here again. Further, if you notice, the IsEmpty method of the Customer class is private. So, in order to use that IsEmpty method inside the CountryMaster class, we need to change the IsEmpty method access specifier to the public as shown in the below image.

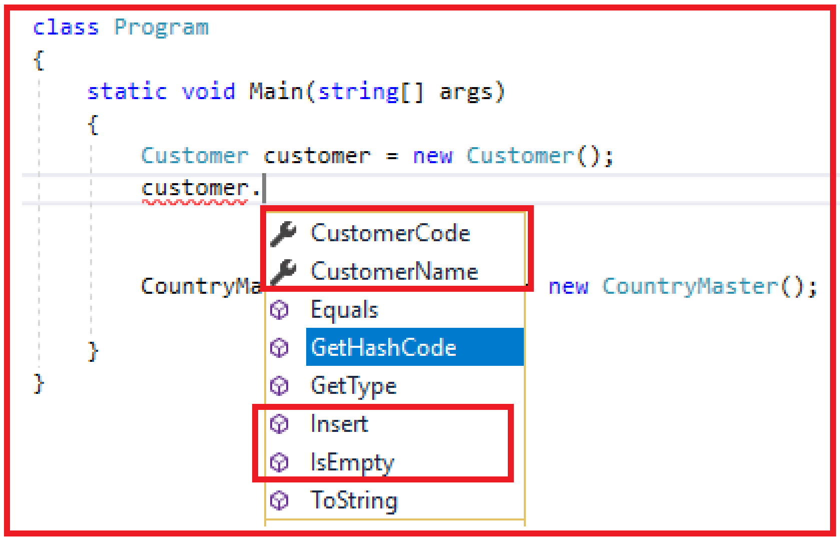


```
public class Customer
{
    public string CustomerCode { get; set; }
    public string CustomerName { get; set; }
    private string MachineName = "";

    public bool IsEmpty(string value)
    {
        if(value.Length > 0)
        {
            return true;
        }

        return false;
    }

    public void Insert()...
}
```

The CountryMaster class has logic to retrieve the computer name and we want to use that logic in the Customer class so we made the ComputerName property public. Similarly, the Customer class has the logic check whether a value is empty or not and we also want that logic in the CountryMaster class, so we made the IsEmpty method as public. As long as we did this, we violate the OOPs principle.

How we are Violating the OOPs Principle?

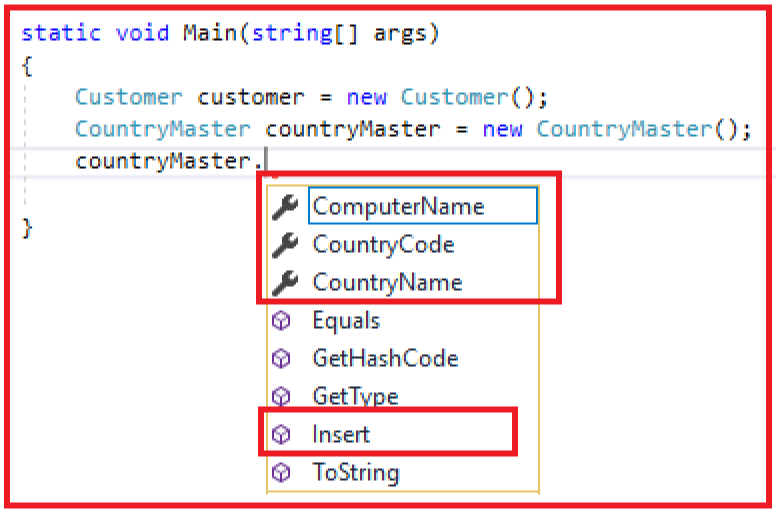
Let us understand how we are violating the OOPs Principle in our code. Now, please modify the Program class as shown below. Once you created the Customer class object, and when you write the object name and dot operator, then the intelligence will show you all the public members of the Customer class as shown in the below image.



```
class Program
{
    static void Main(string[] args)
    {
        Customer customer = new Customer();
        customer.

        CountryMa                        new CountryMaster();

    }
}
```

As you can see in the above image, we have exposed the CustomerCode, CustomerName, Insert, and IsEmpty methods. There is a clear violation of the Abstraction OOPs Principle. Abstraction means showing only what is necessary. So, the external person who is consuming your class, should see and consume the CustomerCode, CustomerName, and Insert method. But should not see the IsEmpty method. The IsEmpty method is for internal use i.e. used by other internal methods of the class and not by the consumer of the class. In this case, the Program class is the consumer of the Customer class i.e. Program class is going to consume the Customer class. As we make the IsEmpty method as public, we are violating the OOPs principle.

In the same way, we are also violating the abstraction principle with the CountryMaster object as we are exposing the ComputerName property to the external world. The ComputerName property is for internal use. That is while inserting the data, it will have the logic to get the computer name and store the same in the database. But, here the consumer of the CountryMaster class can also access and set and get the ComputerName property as shown in the below image. The ComputerName property is for internal use only.



```
static void Main(string[] args)
{
    Customer customer = new Customer();
    CountryMaster countryMaster = new CountryMaster();
    countryMaster.
}
```

Note: With the above, we are achieving code reusability (reusing the ComputerName and IsEmpty method) but violating the OOPS principle.

How to solve the above problem?

How to solve the above problem means how we achieve code reusability without violating the OOPs principles. In order to achieve both, let us add a new class and then move those two functions into that class. Create a class file with the name **CommonTask.cs** and then copy and paste the following code into it.

```
namespace StaticKeyowrdDemo{
    public class CommonTask{
        public bool IsEmpty(string value){
            if (value.Length > 0){
                return true;
            }
            return false;
        }
        public string GetComputerName(){
            return System.Environment.MachineName;
        }
    }
}
```

Now, please remove the IsEmpty() method from the Customer class and the ComputerName property from the CountryMaster class. Now both the logic which violates the OOPs principle has been moved to the **CommonTask** class.

Modifying Customer Class:

Now modify the Customer class as shown below. As you can see, in the constructor, we create an instance of the CommonTask class, and then we set the value of the MachineName private variable. And inside the Insert method, we create an instance of CommonTask class and Invoke the IsEmpty method.

```
namespace StaticKeyowrdDemo{

   public class Customer{

      public string CustomerCode { get; set; }

      public string CustomerName { get; set; }

      private string MachineName = "";

      public Customer(){

         CommonTask commonTask = new CommonTask();

         MachineName = commonTask.ComputerName;

      }

      public void Insert(){

         CommonTask commonTask = new CommonTask();

         if (!commonTask.IsEmpty(CustomerCode) &&
!commonTask.IsEmpty(CustomerName)){

            //Insert the data

         }

      }

   }

}
```

Modifying the CountryMaster Class:

Please modify the CountryMaster class as shown below. Here, we created the instance of CommonTask and then Invoke the ComputerName Property and IsEmpty methods.

```
namespace StaticKeyowrdDemo{
    public class CountryMaster{
        public string CountryCode { get; set; }
        public string CountryName { get; set; }
        private string ComputerName{
            get{
                CommonTask commonTask = new CommonTask();
                return commonTask.ComputerName;
            }
        }
        public void Insert(){
            CommonTask commonTask = new CommonTask();
            if (!commonTask.IsEmpty(CountryCode) && !commonTask.IsEmpty(CountryName)){
                //Logic to Insert the Country Details into the Database
                //ComputerName property tells from which computer the Record is being Inserted
            }
        }
    }
}
```

As we centralized the IsEmpty method and ComputerName property in the CommonTask class, we can use this property and method in both the Customer and CountryMaster classes. The above solution seems to be decent as it does not violate the OOPs Principle and also achieves code reusability and I hope many of you also agree to it. But there is also some problem.

What is the problem in the above solution?

In order to understand the problem, let us first analyze the CommonTask class in a great manner. Please have a look at the following points about the CommonTask class.

This CommonTask class is a collection of unrelated methods and properties that are not related to each other. Because it has unrelated methods, properties, or logic, it does not represent any real-world objects.

As it does not represent any real-world objects, so any kind of OOPs principles (inheritance, abstraction, polymorphism, encapsulation) should not be applied to this CommonTask class.

So, in simple words, we can say that this is a fixed class i.e. a class with a fixed behavior. That is, its behavior cannot be changed by inheritance, and its behavior cannot be polymorphism by using either static or dynamic polymorphism. So, we can say that this class is a fixed class or static class.

How do we avoid Inheritance, how do we avoid abstraction, or how do we avoid the OOPs principle in a class?

The answer is by using the static keyword. So, what we need to do is, we need to mark the CommonTask class as static by using the static keyword. When we mark a class as static, then everything inside the class should also be static. That means, along with the class CommonTask, we also need to mark the IsEmpty method and ComputerName property as static. So, modify the CommonTask class as shown below.

```
namespace StaticKeyowrdDemo{
    public static class CommonTask{
        public static bool IsEmpty(string value){
            if (value.Length > 0){
                return true;
            }
            return false;
        }
        public static string ComputerName{
            get{
                return System.Environment.MachineName;
            }
        }
    }
}
```

Once you make the class static, then you cannot apply any kind of OOPs Principles even you cannot use the **new** keyword with the static class to create an instance rather you need to invoke the **IsEmpty** method and **ComputerName** property by using the class name directly. Internally only one instance of the static class gets created by CLR as soon as the class execution starts and the same single instance will be served by all clients.

Modify the Customer Class:

Now modify the Customer class as shown below. As you can see, now we are invoking the **ComputerName** property and **IsEmpty** method using the class name i.e. **CommonTask** directly without creating any instance.

```
namespace StaticKeyowrdDemo{
    public class Customer{
        public string CustomerCode { get; set; }
        public string CustomerName { get; set; }
        private string MachineName = "";
        public Customer(){
            MachineName = CommonTask.GetComputerName();
        }
        public void Insert(){
            if(!CommonTask.IsEmpty(CustomerCode) &&
!CommonTask.IsEmpty(CustomerName)){
                //Insert the data
            }
        }
    }
}
```

Modify the CountryMaster class:

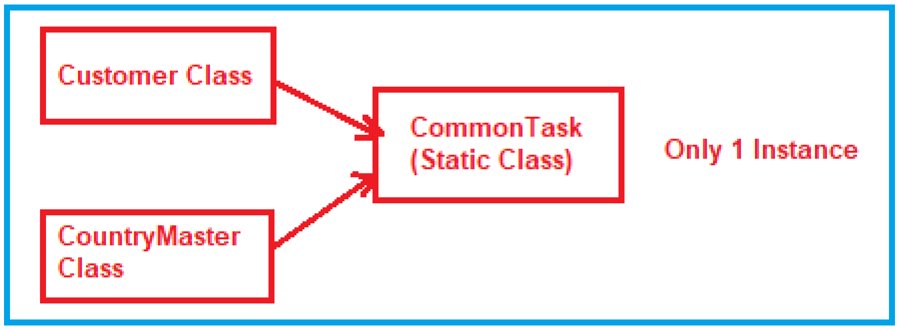
Modify the **CountryMaster** class as shown below. As you can see in the below code, we are invoking the **ComputerName** property and **IsEmpty** method using the class name i.e. **CommonTask** directly without creating any instance.

```
namespace StaticKeyowrdDemo{
    public class CountryMaster{
        public string CountryCode { get; set; }
        public string CountryName { get; set; }
        private string ComputerName{
            get{
                return CommonTask.GetComputerName();
            }
        }
        public void Insert(){
            if (!CommonTask.IsEmpty(CountryCode) && !CommonTask.IsEmpty(CountryName)){
                //Insert the data
            }
        }
    }
}
```

How is the Static Class Instantiated in C#?

We cannot apply any OOPs principles to the static class like inheritance, polymorphism, encapsulation, and abstraction. But in the end, it is a class. And at least to use a class it has to be instantiated. Why because once it is instantiated. then only the static members get memory allocation. Until and unless the memory is not allocated, we cannot access them. So, if the static class is not instantiated then we cannot invoke the methods and properties that are present inside the static class. Now let us see how the instantiation takes place internally of a static class i.e. in our example, it is the **CommonTask** class.

The CLR (Common Language Runtime) internally will create only one instance of the **CommonTask** class irrespective of how many times they called from the **Customer** and **CountryMaster** class. And it is going to be created for the first time when we use consume the **CommonTask** class. For a better understanding, please have a look at the below image.



Due to the single instance behavior, the static class is also going to be used to share the common data.