Encapsulation in C#

Back to: C#.NET Tutorials For Beginners and Professionals

Encapsulation in C# with Examples

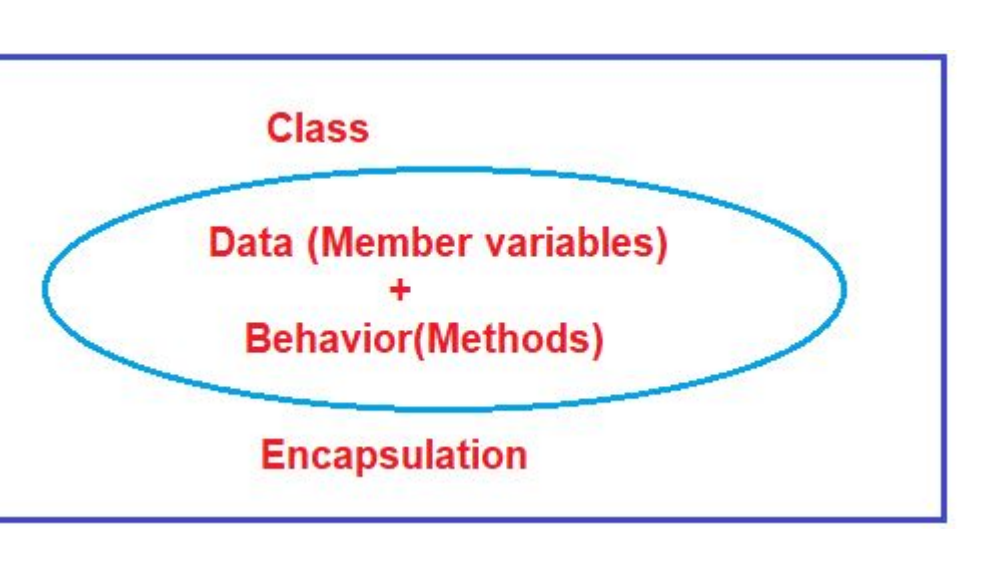
In this article, I will discuss **Encapsulation in C#** with Examples. Please read our previous article before proceeding to this article, where we discussed the **Access Specifies in C#** with Examples. Encapsulation is one of the fundamental principles of Object-Oriented Programming. Many Students and Programmers, especially beginners, find it a little difficult to understand what exactly the Encapsulation Principle is. So, in this article, I will try to make it as simple as possible so that you can understand it easily. As part of this article, you will understand the following pointers in detail.

1. **What is Encapsulation?**
2. **How can we Implement Encapsulation in C#?**
3. **Encapsulation Examples Using Setter and Getter Methods**
4. **What are the Advantages of Providing Variable Access via Setter and Getter Methods in C#?**
5. **What is the Problem if we don't follow the Encapsulation Principle in C# while Designing a Class?**
6. **Implementing Encapsulation in C# Using Properties**

What is the Encapsulation Principle in C#?

According to MSDN, **Encapsulation Hides the internal state and functionality of an object and only allows access through a public set of functions**. Let us simplify the above definition as follows:

The process of binding or grouping the **State (i.e., Data Members)** and **Behaviour (i.e., Member Functions)** together into a single unit (i.e., **class, interface, struct, etc.**) is called Encapsulation in C#. The Encapsulation Principle ensures that the state and behavior of a unit (i.e., **class, interface, struct, etc.**) cannot be accessed directly from other units (i.e., **class, interface, struct, etc.**).



The Encapsulation Principle in C# is very similar to a Capsule. As a capsule binds its medicine within it, in the same way in C#, the Encapsulation Principle binds the State (Variables) and Behaviour (Methods) into a single unit called class, enum, interface, etc. So, you can think of Encapsulation as a cover or layer that binds related states and behavior together in a single unit.

Real-World Example of Encapsulation:

As we already discussed, one of the real-world examples of encapsulation is the Capsule, as the capsule binds all its medicinal materials within it. In the same way, C# Encapsulation, i.e., units (class, interface, enums, structs, etc) encloses all its data member and member functions within it.

Another real-world example of encapsulation can be your school or office bag. The bag contains different stuff like a Pen, Pencil, Notebook, etc it. To get any stuff, you need to open that bag. Similarly, in C#, an encapsulation unit contains its data and behavior within it, and in order to access them, you need an object of that unit.

Example to Understand Encapsulation in C#:

Every class, interface, struct, enum, etc. that we created is an example of encapsulation, so let's create a class called Bank as follows to understand the encapsulation:

```
namespace EncapsulationDemo
{
    class Bank
    {
        public long AccountNumber;
        public string Name;
        public int Balance;

        public void GetBalance()
        {
        }
        public void WithdrawAmount()
        {
        }
        public void Deposit()
        {
        }
    }
}
```

Here, the class Bank is an example of Encapsulation. The variables(AccountNumber, Name, and Balance) and methods(GetBalance, WithdrawAmount, and Deposit) of the class are bound in a single unit, which is the Bank class. Here, the encapsulation binds the implementation details of the Bank class with it and hides it from other classes. If other classes want to access these details, they need to create the object of the Bank class to access its data and behavior, as shown in the code below.

```
namespace EncapsulationDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            Bank bank = new Bank();
            bank.AccountNumber = 12345678;
            bank.Name = "Pranata";
            bank.GetBalance();
            bank.WithdrawAmount();
        }
    }
}
```

Similarly, if you create an interface, struct, or enum, that is also an example of encapsulation. The biggest advantage of Encapsulation is Data Hiding. That means, by using Encapsulation, we can achieve Data Hiding. Let us proceed further and understand Data Hiding in C# with Examples.

What is Data Hiding in C#?

Data hiding or Information Hiding is a Process in which we hide internal data from outside the world. The purpose of data hiding is to protect the data from misuse by the outside world. Data hiding is also known as Data Encapsulation. Without the Encapsulation Principle, we cannot achieve data hiding.

In simple words, we can also say that the process of defining a class by hiding its internal data members from outside the class and accessing those internal data members only through publicly exposed methods (setter and getter methods) or properties with proper validations is called Encapsulation.

Note: Data Encapsulation is also called Data Hiding because by using this principle, we can hide the internal data from outside the class.

How can we Implement Data Hiding or Data Encapsulation in C#?

In C#, Data Encapsulation is implemented.

1. By declaring the variables as private (to restrict their direct access from outside the class)
2. By defining one pair of public setter and getter methods or properties to access private variables from outside the class.

We declare variables as private to stop accessing them directly from outside the class. The public setter and getter methods or public properties are used to access the private variables from outside the class with proper validations. If we provide direct access to the variables of a class, then we cannot validate the data before storing it in the variable or while retrieving the data stored in the variable. So the point that you need to remember is by implementing Data Encapsulation or Data Hiding in C#, we are protecting or securing the data.

So, Data Encapsulation or Data Hiding in C# is implemented by using the **Access Specifiers**. An access specifier defines the scope and visibility of the class member, and we have already discussed the different types of Access Specifiers Supported in C# in our previous article. C# supports the following six access specifiers:

1. **public**: The public members can be accessed by any other code in the same assembly or another assembly that references it.
2. **private**: The private members can be accessed only by code in the same class.
3. **protected**: The protected Members in C# are available within the same class as well as to the classes that are derived from that class.
4. **internal**: The internal members can be accessed by any code in the same assembly but not from another assembly.
5. **protected internal**: The protected internal members can be accessed by any code in the assembly in which it's declared or from within a derived class in another assembly.
6. **private protected:** The private protected members can be accessed by types derived from the class that is declared within its containing assembly.

Implementing Data Encapsulation or Data Hiding in C# using Setter and Getter Methods:

Let us see an example to understand Encapsulation in C#. In the following example, we declare the balance variable as private in the Bank class, and hence, it can not be accessed directly from outside of the Bank class. To access the private balance variable from outside the Bank class, we have exposed two public methods, i.e., GetBalance and SetBalance. The GetBalance method (which is also called getter) is used to fetch the value stored in the private balance variable, and the SetBalance method (which is also called Setter) is used to set the value in the private balance variable from outside the bank class. Within the Bank class, you can access the private variables directly, but you cannot access them directly from outside of the Bank class.

```
using System;
namespace EncapsulationDemo
{
    public class Bank
    {
        //Hiding class data by declaring the variable as private
        private double balance;

        //Creating public Setter and Getter methods

        //Public Getter Method
        //This method is used to return the data stored in the balance variable
        public double GetBalance()
        {
            //add validation logic if needed
            return balance;
        }

        //Public Setter Method
        //This method is used to stored the data  in the balance variable
        public void SetBalance(double balance)
        {
            // add validation logic to check whether data is correct or not
            this.balance = balance;
        }
    }
    class Program
    {
        public static void Main()
        {
            Bank bank = new Bank();
            //You cannot access the Private Variable
            //bank.balance; //Compile Time Error

            //You can access the private variable via public setter and getter methods
            bank.SetBalance(500);
            Console.WriteLine(bank.GetBalance());
            Console.ReadKey();
        }
    }
}
```

Output: 500

What are the Advantages of Providing Variable Access via Setter and Getter Methods in C#?

If we provide Variable Access via Setter and Getter Methods in C#, we can validate the user-given data before storing the value in the variable. In the above program, for example, if you don't want to store the -VE value in the balance variable, you check and validate the value before storing it in the variable. So, we can validate the given value before storing it in the balance variable. If we provide direct access to the balance variable, it is impossible to validate the given amount value before storing it in the balance variable.

So, the main reason for using data hiding is security. As we use private access specifiers with our variables, we can store critical information in such variables, which will only be visible within the class. No one else can access them directly. We can also apply some validation in setter and getter methods whenever needed. It also increases security so that no one can set any illegal data for misuse.

What is the Problem if we don't follow the Encapsulation Principle in C# while Designing a Class?

If we don't follow the Encapsulation Principle in C# while designing the class, we cannot validate the user-given data according to our business requirements, as it is very difficult to handle future changes. Let us understand this with an example. Assume in the initial project requirement, the client did not mention that the application should not allow the negative number to be stored. So, we give direct access to the variable from outside the class, and now, the user can store any value to it, as shown in the below example. Here, you can see that we are accessing the Amount variable directly from outside the Bank class and setting both positive and negative values into it.

```
using System;
namespace EncapsulationDemo
{
    public class Bank
    {
        public int Amount;
    }
    class Program
    {
        public static void Main()
        {
            Bank bank = new Bank();
            //We can access the Amount Variable directly

            //Setting positive amount
            bank.Amount = 50;
            Console.WriteLine(bank.Amount);

            //Setting negative amount
            bank.Amount = -150;
            Console.WriteLine(bank.Amount);

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

Output:

50
-150

That's it. It works as expected. Later, in the future, the client wants that the application should not allow a negative value. Then, we should validate the user-given values before storing them in the Amount variable. Hence, we need to develop the application by following the Encapsulation Principle as follows:

```
using System;
namespace EncapsulationDemo
{
    public class Bank
    {
        private int Amount;
        public int GetAmount()
        {
            return Amount;
        }
        public void SetAmount(int Amount)
        {
            if (Amount > 0)
            {
                this.Amount = Amount;
            }
            else
            {
                throw new Exception("Please Pass a Positive Value");
            }
        }
    }
    class Program
    {
        public static void Main()
        {
            try
            {
                Bank bank = new Bank();
                //We cannot access the Amount Variable directly
                //bank.Amount = 50; //Compile Time Error
                //Console.WriteLine(bank.Amount); //Compile Time Error

                //Setting Positive Value
                bank.SetAmount(10);
                Console.WriteLine(bank.GetAmount());

                //Setting Negative Value
                bank.SetAmount(-150);
                Console.WriteLine(bank.GetAmount());
            }
            catch(Exception ex)
            {
                Console.WriteLine(ex.Message);
            }

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

Output:

```
10
Please Pass a Positive Value
```

Implementing Data Encapsulation or Data Hiding in C# using Properties:

The Properties are a new language feature introduced in C#. Properties in C# help in protecting a field or variable of a class by reading and writing the values to it. The first approach, i.e., setter and getter itself, is good, but Data Encapsulation in C# can be accomplished much smoother with properties.

Let us understand how to implement Data Encapsulation or Data Hiding in C# using properties with an example. In the below example, inside the Bank class, we marked the _Amount variable as private to restrict direct access from outside the Bank class. We have exposed the Amount property to access the _Amount variable by declaring it as public. Now, from outside the Bank class, we can access the _Amount private variable through the public exposed Amount property.

```
using System;
namespace EncapsulationDemo
{
    public class Bank
    {
        private double _Amount;
        public double Amount
        {
            get
            {
                return _Amount;
            }
            set
            {
                // Validate the value before storing it in the _Amount variable
                if (value < 0)
                {
                    throw new Exception("Please Pass a Positive Value");
                }
                else
                {
                    _Amount = value;
                }
            }
        }
    }
    class Program
    {
        public static void Main()
        {
            try
            {
                Bank bank = new Bank();
                //We cannot access the _Amount Variable directly
                //bank._Amount = 50; //Compile Time Error
                //Console.WriteLine(bank._Amount); //Compile Time Error

                //Setting Positive Value using public Amount Property
                bank.Amount= 10;

                //Setting the Value using public Amount Property
                Console.WriteLine(bank.Amount);

                //Setting Negative Value
                bank.Amount = -150;
                Console.WriteLine(bank.Amount);
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }

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

Output:

```
10
Please Pass a Positive Value
```

Advantages of Encapsulation in C#:

1. **Data protection:** You can validate the data before storing it in the variable.
2. **Achieving Data Hiding:** The user will have no idea about the inner implementation of the class.
3. **Security:** The encapsulation Principle helps to secure our code since it ensures that other units(classes, interfaces, etc) can not access the data directly.
4. **Flexibility:** The encapsulation Principle in C# makes our code more flexible, allowing the programmer to easily change or update the code.
5. **Control:** The encapsulation Principle gives more control over the data stored in the variables. For example, we can control the data by validating whether the data is good enough to store in the variable.

Encapsulation is one of the four fundamental principles of Object-Oriented Programming (OOP). It refers to the bundling of data (state) and methods (behaviors) that operate on the data into a single unit and restricting access to some of the object's components.

In the next article, I will discuss **Abstraction in C#** with Examples. In this article, I try to explain **Encapsulation in C#** with Examples. I hope this article will help you with your needs. I would like to have your feedback. Please post your feedback, questions, or comments about this Encapsulation in C# article.

Dot Net Tutorials
About the Author: Pranaya Rout
Pranaya Rout has published more than 3,000 articles in his 11-year career. Pranaya Rout has very good experience with Microsoft Technologies, Including C#, VB, ASP.NET MVC, ASP.NET Web API, EF, EF Core, ADO.NET, LINQ, SQL Server, MYSQL, Oracle, ASP.NET Core, Cloud Computing, Microservices, Design Patterns and still learning new technologies.

Previous Lesson
Access Specifiers in C#

Next Lesson
Abstraction in C#

10 thoughts on "Encapsulation in C#"

LOIS
MAY 18, 2020 AT 2:50 AM

Thank you so much for this enlightening and interesting explanation of encapsulation.

Reply

PADAM
JUNE 21, 2020 AT 8:06 PM

Thank you for such as informative details.

Reply

KUSHAL
DECEMBER 29, 2020 AT 11:07 PM

good explanation.

Reply

SUDHA
MARCH 23, 2021 AT 10:45 AM

Thank you for the great explanation. But I have read in stack overflow that private double balance; is not a variable but a field

Reply

ANURAG MOHANTY
MAY 1, 2021 AT 1:29 PM

very nice

Reply

SAILAJA BATCHANABOINA
DECEMBER 30, 2021 AT 2:07 PM

ThankYou so much.its a nice explanation.

Reply

RAVEENA
FEBRUARY 22, 2022 AT 3:25 AM

Great tutorials, but in the final output, how did it end up being -50 when in the setter we have defined negative values aren't allowed. When I run it locally, the 100 value persists.

Reply

ARAVINDH
MAY 18, 2022 AT 8:19 AM

Hi Raveena,

Yes, your suggestion is correct.

Regards,
Aravindh B

Reply

MICHAEL
OCTOBER 3, 2022 AT 9:45 PM

Now I have understood the advantage of get and set, thank you very much.

Reply

CRIMSON206
DECEMBER 14, 2022 AT 9:21 PM

Even if we don't need a validation yet, do we need to encapsulate data?
public int Balance { get; set; }
is short enough to replace
public int Balance;

However; it is redundant to write both field and property for the time being, ex :
private int balance;
public int Balance { get => this.balance; set => this.balances = value }

1. I need to write one more line for nothing.
2. Readers also need to read one more useless line.
3. If I want to add a validation line, I usually need to rewrite them thoroughly because lambda function expression is not for complex implementation. Therefore, I can't even reuse the property later. ex)

private int balance;
public int Balance
{
get { return this.balance };
set
{
if (value < 0) throw new Exception();
this.balance = value;
}
}

Reply

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment *

Name* | Email* | Website

Post Comment