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 According to MSDN, Encapsulation Hides the internal state and functionality of an object and only allows access 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, 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 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 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 Every class, interface, struct, enum, etc. that we created is an example of encapsulation, so let's create a class called Bank as 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 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 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 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 Note: Data Encapsulation is also called Data Hiding because by using this principle, we can hide the internal data from **Application** 2. By defining one pair of public setter and getter methods or properties to access private variables from outside the class. ✓ Interface in C# Interface Interview Questions We declare variables as private to stop accessing them directly from outside the class. The public setter and getter methods or and Answers in C# public properties are used to access the private variables from outside the class with proper validations. If we provide direct Interface Realtime Examples access to the variables of a class, then we cannot validate the data before storing it in the variable or while retrieving the data in C# stored in the variable. So the point that you need to remember is by implementing Data Encapsulation or Data Hiding in C#, Multiple Inheritance in C# we are protecting or securing the data. Multiple Inheritance Realtime Example in C# So, Data Encapsulation or Data Hiding in C# is implemented by using the Access Specifiers. An access specifier defines the Polymorphism in C# scope and visibility of the class member, and we have already discussed the different types of Access Specifiers Supported in Method Overloading in C# C# in our previous article. C# supports the following six access specifiers: Operator Overloading in C# Method Overriding in C# 1. public: The public members can be accessed by any other code in the same assembly or another assembly that Method Hiding in C# references it. Partial Class and Partial 2. **private**: The private members can be accessed only by code in the same class. Methods in C# 3. protected: The protected Members in C# are available within the same class as well as to the classes that are derived Sealed Class and Sealed from that class. Methods in C# 4. **internal**: The internal members can be accessed by any code in the same assembly but not from another assembly. Extension Methods in C# 5. protected internal: The protected internal members can be accessed by any code in the assembly in which it's Static Class in C# declared or from within a derived class in another assembly. ✓ Variable Reference and 6. **private protected:** The private protected members can be accessed by types derived from the class that is declared Instance of a Class in C# within its containing assembly. Real-time Examples of Implementing Data Encapsulation or Data Hiding in C# using Setter and Getter Methods: Encapsulation Principle in C# Real-Time Examples of Let us see an example to understand Encapsulation in C#. In the following example, we declare the balance variable as Abstraction Principle in C# private in the Bank class, and hence, it can not be accessed directly from outside of the Bank class. To access the private Real-Time Examples of balance variable from outside the Bank class, we have exposed two public methods, i.e., GetBalance and SetBalance. The Inheritance Principle in C# GetBalance method (which is also called getter) is used to fetch the value stored in the private balance variable, and the Real-Time Examples of SetBalance method (which is also called Setter) is used to set the value in the private balance variable from outside the bank Polymorphism Principle in C# class. Within the Bank class, you can access the private variables directly, but you cannot access them directly from outside of Real-Time Examples of the Bank class. Interface in C# Real-Time Examples of using System; Abstract Class in C# namespace EncapsulationDemo **Exception Handling** public class Bank Exception Handling in C# //Hiding class data by declaring the variable as private ✓ Multiple Catch Blocks in C# private double balance; Finally Block in C# //Creating public Setter and Getter methods How to Create Custom Exceptions in C# //Public Getter Method ✓ Inner Exception in C# //This method is used to return the data stored in the balance variable public double GetBalance() Exception Handling Abuse in C# //add validation logic if needed return balance; Events, Delegates and Lambda Expression in C# //Public Setter Method Course Structure of Events. //This method is used to stored the data in the balance variable Delegates and Lambda public void SetBalance(double balance) Expression Roles of Events, Delegates // add validation logic to check whether data is correct or not this.balance = balance; and Event Handler in C# Delegates in C# Multicast Delegates in C# class Program Delegates Real-Time Example public static void Main() in C# Generic Delegates in C# Bank bank = new Bank(); //You cannot access the Private Variable Anonymous Method in C# //bank.balance; //Compile Time Error Lambda Expressions in C# Events in C# with Examples //You can access the private variable via public setter and getter methods bank.SetBalance(500); Console.WriteLine(bank.GetBalance()); Console.ReadKey(); Multithreading in C# Thread class in C# How to Pass Data to Thread Function in Type Safe Manner Output: 500 in C# How to Retrieve Data from a What are the Advantages of Providing Variable Access via Setter and Getter Methods in Thread Function in C# C#? Join Method and IsAlive Property of Thread Class in 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 ▼ Thread Synchronization in C# check and validate the value before storing it in the variable. So, we can validate the given value before storing it in the Lock in C# balance variable. If we provide direct access to the balance variable, it is impossible to validate the given amount value before Monitor Class in C# storing it in the balance variable. Mutex Class in C# Semaphore Class in C# So, the main reason for using data hiding is security. As we use private access specifiers with our variables, we can store SemaphoreSlim Class in C# critical information in such variables, which will only be visible within the class. No one else can access them directly. We can Deadlock in C# also apply some validation in setter and getter methods whenever needed. It also increases security so that no one can set Performance Testing of a Multithreaded Application any illegal data for misuse. Thread Pool in C# Foreground and Background What is the Problem if we don't follow the Encapsulation Principle in C# while Designing a Class? Threads in C# If we don't follow the Encapsulation Principle in C# while designing the class, we cannot validate the user-given data AutoResetEvent and according to our business requirements, as it is very difficult to handle future changes. Let us understand this with an example. ManualResetEvent in C# ▼ Thread Life Cycle in C# Assume in the initial project requirement, the client did not mention that the application should not allow the negative number ▼ Threads Priorities in C# 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 How to Terminate a Thread in shown in the below example. Here, you can see that we are accessing the Amount variable directly from outside the Bank C# class and setting both positive and negative values into it. Inter Thread Communication in C# using System; namespace EncapsulationDemo How to Debug a Multithreaded Application in C# public class Bank public int Amount; Arrays in C# class Program 2D Arrays in C# Advantages and public static void Main() Disadvantages of Arrays in C# Bank bank = new Bank(); Collections in C# //We can access the Amount Variable directly ArrayList in C# //Setting positive amount ✓ Hashtable in C# bank.Amount = 50; Non-Generic Stack in C# Console.WriteLine(bank.Amount); Non-Generic Queue in C# Non-Generic SortedList in C# bank.Amount = -150; Advantages and Console.WriteLine(bank.Amount); Disadvantages of Non-Console.ReadKey(); Generic Collection in C# Generic Collections in C# Generics in C# Generic Constraints in C# **Output:** Generic List Collection in C# How to Sort a List of **50** Complex Type in C# -150 Comparison Delegate in C# Dictionary Collection Class in That's it. It works as expected. Later, in the future, the client wants that the application should not allow a negative value. C# Then, we should validate the user-given values before storing them in the Amount variable. Hence, we need to develop the Conversion Between Array application by following the Encapsulation Principle as follows: List and Dictionary in C# List vs Dictionary in C# using System; Generic Stack Collection namespace EncapsulationDemo Class in C# public class Bank Generic Queue Collection Class in C# private int Amount; Foreach Loop in C# public int GetAmount() Generic HashSet Collection return Amount; Class in C# public void SetAmount(int Amount) Class in C# if (Amount > 0) Class in C# this.Amount = Amount; Generic SortedDictionary else Collection Class in C# throw new Exception("Please Pass a Positive Value"); Class in C# Concurrent Collection in C# ConcurrentDictionary class Program Collection Class in C# public static void Main() ConcurrentQueue Collection Class in C# try ConcurrentStack Collection Class in C# Bank bank = new Bank(); ConcurrentBag Collection Class in C# //Console.WriteLine(bank.Amount); //Compile Time Error BlockingCollection in C# bank.SetAmount(10); Console.WriteLine(bank.GetAmount()); File Handling in C# FileStream Class in C# StreamReader and bank.SetAmount(-150); Console.WriteLine(bank.GetAmount()); StreamWriter in C# File Class in C# catch(Exception ex) TextWriter and TextReader in Console.WriteLine(ex.Message); BinaryWriter and BinaryReader in C# Console.ReadKey(); StringWriter and StringReader in C# FileInfo Class in C# DirectoryInfo Class in C# **Output:** Export and Import Excel Data in C# Please Pass a Positive Value Introduction to Concurrency Implementing Data Encapsulation or Data Hiding in C# using Properties: Async and Await in C# The Properties are a new language feature introduced in C#. Properties in C# help in protecting a field or variable of a class by Task in C# reading and writing the values to it. The first approach, i.e., setter and getter itself, is good, but Data Encapsulation in C# can How to Return a Value from Task in C# be accomplished much smoother with properties. How to Execute Multiple Let us understand how to implement Data Encapsulation or Data Hiding in C# using properties with an example. In the below Tasks in C# How to Limit Number of example, inside the Bank class, we marked the \_Amount variable as private to restrict direct access from outside the Bank Concurrent Tasks in C# class. We have exposed the Amount property to access the \_Amount variable by declaring it as public. Now, from outside the How to Cancel a Task in C# Bank class, we can access the \_Amount private variable through the public exposed Amount property.

> class Program public static void Main() try Bank bank = new Bank(); //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:** Please Pass a Positive Value Advantages of Encapsulation in C#: 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. in X 🖸 🕲 🕢 **Previous Lesson** Next Lesson Access Specifiers in C# 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.

**PADAM** 

**KUSHAL** 

**SUDHA** 

good explanation.

JUNE 21, 2020 AT 8:06 PM

Thank you for such as informative details.

DECEMBER 29, 2020 AT 11:07 PM

MARCH 23, 2021 AT 10:45 AM

**ANURAG MOHANTY** 

MAY 1, 2021 AT 1:29 PM

SAILAJA BATCHANABOINA

ThankYou so much.its a nice explanation.

**RAVEENA** 

DECEMBER 30, 2021 AT 2:07 PM

FEBRUARY 22, 2022 AT 3:25 AM

**ARAVINDH** 

get { return this.balance };

this.balance = value;

if (value < 0 ) throw new Exception();

set

allowed. When I run it locally, the 100 value persists.

Reply

Reply

Reply

Reply

Reply

Reply

Reply

Reply

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

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

Name\* Email\* Website **Post Comment** 

About Us Privacy Policy Contact ADO.NET Tutorial Angular Tutorials ASP.NET Core Blazor Tuturials ASP.NET Core Tutorials ASP.NET MVC Tutorials ASP.NET Web API Tutorials C Tutorials C#.NET Programs Tutorials C#.NET Tutorials Cloud Computing Tutorials Data Structures and Algorithms Tutorials Design Patterns Tutorials DotNet Interview Questions and Answers Core Java Tutorials Entity Framework Tutorials JavaScript Tutorials LINQ Tutorials Python Tutorials SOLID Principles Tutorials SQL Server Tutorials Trading Tutorials JDBC Tutorials Java Servlets Tutorials Java Struts Tutorials C++ Tutorials JSP Tutorials MySQL Tutorials Oracle Tutorials ASP.NET Core Web API Tutorials HTML Tutorials

Types in C# Stackalloc in in C# Most Popular C# Books Most Recommended C# **Books** 

Comment \*

OOPs Real-Time Examples

**Multi-Threading** 

Collections in C#

**Asynchronous Programming** using Cancellation Token How to Create Synchronous Method using Task in C# Retry Pattern in C# Only One Pattern in C# How to Control the Result of a Task in C# ▼ Task-Based Asynchronous

Programming in C#

Chaining Tasks by Using Continuation Tasks

to a Parent Task in C#

✓ ValueTask in C#

How to Attached Child Tasks

Primitive Type using AutoMapper in C# AutoMapper Reverse Mapping in C# AutoMapper Conditional Mapping in C# AutoMapper Ignore Method in Fixed and Dynamic Values in **Destination Property in** AutoMapper Optional Parameter, Indexers and Enums How to make Optional Parameters in C# ✓ Indexers in C# Indexers Real-Time Example in C# Enums in C# .NET Framework Architecture DOT NET Framework Common Language Runtime in .NET Framework .NET Program Execution Process Intermediate Language (ILDASM & ILASM) Code in Common Type System in .NET Framework Common Language

Specification in .NET

Managed and Unmanaged

Code in .NET Framework

Strong and Weak Assemblies

How to Install an Assembly

into GAC in .NET Framework

Solution in .NET Framework

Assembly DLL EXE in .NET

Framework

Framework

Framework

App Domain in .NET

in .NET Framework

DLL Hell Problem and

Var, Dynamic and Reflection

Dynamic Type in C#

✓ Var Keyword in C#

✓ Var vs Dynamic in C#

∇ Volatile Keyword in C#

Named Parameters in C#

Enhancement in Out Variables

Ref vs Out in C#

C# 7.X new Features

C# 7 New Features

Dynamic vs Reflection in C#

Reflection in C#

Async Main in C# C# 8 New Features C# 8 New Features ReadOnly Structs in C# Default Interface Methods in C# Pattern Matching in C# Using Declarations in C# Static Local Functions in C# Disposable Ref Structs in C# Nullable Reference Types in C#8 Asynchronous Streams in C# Asynchronous Disposable in Indices and Ranges in C# Null-Coalescing Assignment Operator in C# Unmanaged Constructed

Most Recommended Data Books using C#

How to Cancel a Non-Cancellable Task in C# Asynchronous Streams in C# How to Cancel Asynchronous Stream in C# Parallel Programming Task Parallel Library in C# Parallel For in C# Parallel Foreach Loop in C# Atomic Methods Thread Safety and Race Conditions in Interlocked vs Lock in C#

using System;

namespace EncapsulationDemo

private double \_Amount;

return \_Amount;

if (value < 0)

\_Amount = value;

throw new Exception("Please Pass a Positive Value");

public double Amount

else

public class Bank

get

set

 ▼ Thrown Expression in C# Thank you for the great explanation. But I have read in stack overflow that private double balance; is not a variable but a field very nice

File Handling

Parallel Invoke in C# Maximum Degree of Parallelism in C# How to Cancel Parallel Operations in C# C# Parallel LINQ in C# Multithreading vs **Asynchronous Programming** vs Parallel Programming in C# AutoMapper AutoMapper in C# AutoMapper Complex Mapping in C# How to Map Complex Type to

in C# 7 Pattern Matching in C# Digit Separators in C# 7 ▼ Tuples in C# 7 Splitting Tuples in C# 7 Local Functions in C# 7 Ref Returns and Ref Locals in C# 7 Generalized Async Return Types in C# 7 Expression Bodied Members in C#

Structure and Algorithms

· © Dot Net Tutorials | Website Design by Sunrise Pixel

Leave a Reply Your email address will not be published. Required fields are marked \*

