**Encapsulation**

Encapsulation in Object-Oriented Programming (OOP) is the concept of bundling data (variables) and methods that operate on the data into a single unit, known as a class. Additionally, it involves restricting access to some of the object's components to prevent unintended interference and misuse of the data. This is done using access modifiers like public, private, protected, and internal.

**Real-Life Example: Bank Account System**

Let's imagine a **Bank Account** system where we want to ensure that the account balance can only be changed in controlled ways (for example, only through depositing or withdrawing money, and not directly altering the balance).

In this scenario, **Encapsulation** is used to:

1. **Hide** the internal state (e.g., account balance) from the outside world.
2. **Control** access to the state using public methods (e.g., Deposit(), Withdraw()).

**Step-by-Step Example of Encapsulation in C#**

**1. BankAccount Class with Encapsulation**

We will create a BankAccount class with:

* A **private** field for the balance, so it cannot be directly modified from outside the class.
* **Public methods** for deposit and withdrawal to control how the balance is updated.

public class BankAccount

{

// Private field: encapsulated data, not directly accessible from outside

private decimal balance;

// Constructor to initialize a new account with a starting balance

public BankAccount(decimal initialBalance)

{

balance = initialBalance;

}

// Public method to deposit money into the account

public void Deposit(decimal amount)

{

if (amount > 0)

{

balance += amount;

Console.WriteLine($"Deposited: {amount}. New balance: {balance}");

}

else

{

Console.WriteLine("Deposit amount must be positive.");

}

}

// Public method to withdraw money from the account

public void Withdraw(decimal amount)

{

if (amount <= 0)

{

Console.WriteLine("Withdrawal amount must be positive.");

return;

}

if (amount <= balance)

{

balance -= amount;

Console.WriteLine($"Withdrew: {amount}. New balance: {balance}");

}

else

{

Console.WriteLine("Insufficient funds.");

}

}

// Public method to check the current balance (read-only access to balance)

public decimal GetBalance()

{

return balance;

}

}

**2. Using the BankAccount Class**

Now, let's create an instance of BankAccount and use its methods to deposit and withdraw money. Notice that we can't directly access the balance field; we can only interact with it via the public methods.

public class Program

{

public static void Main(string[] args)

{

// Create a new BankAccount object with an initial balance of 1000

BankAccount account = new BankAccount(1000);

// Deposit money into the account

account.Deposit(500); // New balance: 1500

// Withdraw money from the account

account.Withdraw(300); // New balance: 1200

// Attempt an invalid withdrawal (more than the balance)

account.Withdraw(1500); // Insufficient funds

// Check balance (read-only access)

Console.WriteLine($"Current Balance: {account.GetBalance()}");

}

}

**Output:**

Deposited: 500. New balance: 1500

Withdrew: 300. New balance: 1200

Insufficient funds.

Current Balance: 1200

**Explanation of Encapsulation in This Example:**

1. **Private Data** (balance):
   * The balance field is marked as private. This means that it cannot be directly accessed or modified from outside the BankAccount class.
   * This ensures that the balance can only be modified in controlled ways, such as through deposits or withdrawals.
2. **Public Methods** (Deposit(), Withdraw(), GetBalance()):
   * These methods provide controlled access to the balance. They are the only way to modify or view the balance.
   * The Deposit() method adds money to the account, while the Withdraw() method deducts money. The withdrawal method includes validation (checking if the amount is valid and if there are sufficient funds) to prevent improper changes to the balance.
   * GetBalance() allows us to check the current balance but doesn't allow direct modification of it.
3. **Access Control**:
   * By making the balance field private, we ensure that other parts of the program cannot access or change the balance directly. This **protects the data** and prevents errors or unintended changes.
   * The public methods are used to ensure that the data is manipulated in a safe and predictable manner.

**Why is Encapsulation Important?**

1. **Control Access**: We control how the data is accessed and modified, ensuring that invalid operations (like depositing negative amounts or withdrawing more than the balance) are prevented.
2. **Data Integrity**: It helps protect the internal state of the object, ensuring that only valid operations can affect the data.
3. **Code Maintainability**: If we need to change the way the balance is calculated or stored, we can do it within the class without affecting the rest of the application. The interface (public methods) remains the same.
4. **Security**: By hiding internal data and exposing only the necessary parts, encapsulation can prevent external code from accidentally or maliciously modifying important data.

**Conclusion**

Encapsulation in C# provides a mechanism for hiding data and controlling access to it. In this **BankAccount** example, we have encapsulated the balance field by making it private, and we control how it is accessed and modified via public methods. This ensures that the account balance is always manipulated in a controlled, valid way, preventing errors and improving the integrity and maintainability of the code.