**Encapsulation**

Encapsulation is one of the fundamental principles of object-oriented programming (OOP), and it is used in C# to restrict access to the internal state of an object and allow manipulation of that state only through well-defined methods. This ensures better control, security, and maintainability of the code.

**Key Concepts of Encapsulation:**

1. **Access Modifiers:** Encapsulation relies on access modifiers (private, public, protected, internal, etc.) to control access to class members.
   * private: Restricts access to the containing class only.
   * public: Makes the member accessible from any other class.
   * protected: Allows access within the containing class and derived classes.
   * internal: Allows access within the same assembly.
2. **Getters and Setters:** Use properties to define controlled access to private fields. Properties allow encapsulation by providing a way to read (get) or modify (set) private fields with additional logic.

In C#, **getters** and **setters** are special methods used in **properties** to control access to the fields of a class. They allow encapsulation by enabling you to get (retrieve) or set (modify) the value of private fields with added logic or restrictions.

**Syntax of Getters and Setters**

public class ClassName

{

// Private field

private DataType fieldName;

// Public property

public DataType PropertyName

{

get { return fieldName; } // Getter: retrieves the field's value

set { fieldName = value; } // Setter: sets the field's value

}

}

1. **Example: Encapsulation in C#**

using System;

class Person

{

// Private fields (encapsulated data)

private string name;

private int age;

// Public property for 'Name' with encapsulated logic

public string Name

{

get { return name; }

set

{

if (!string.IsNullOrEmpty(value))

name = value;

else

Console.WriteLine("Name cannot be empty.");

}

}

// Public property for 'Age' with encapsulated logic

public int Age

{

get { return age; }

set

{

if (value >= 0)

age = value;

else

Console.WriteLine("Age cannot be negative.");

}

}

// Constructor

public Person(string name, int age)

{

Name = name; // Uses the setter

Age = age; // Uses the setter

}

// Method to display details (controlled access)

public void DisplayDetails()

{

Console.WriteLine($"Name: {Name}, Age: {Age}");

}

}

class Program

{

static void Main()

{

// Creating an object of the Person class

Person person = new Person("Alice", 25);

// Accessing encapsulated data via properties

Console.WriteLine(person.Name); // Output: Alice

person.Age = 30; // Updates the age using the setter

person.DisplayDetails(); // Output: Name: Alice, Age: 30

// Attempting invalid assignment

person.Age = -5; // "Age cannot be negative."

}

}

**Benefits of Encapsulation:**

1. **Data Hiding:** Prevents unauthorized access and modification of object data.
2. **Flexibility:** Allows changes to internal implementation without affecting external code.
3. **Control:** Enables validation or additional logic when accessing or modifying data.
4. **Maintainability:** Encapsulation makes it easier to manage and debug the code.

Encapsulation enforces the idea of separating "what an object does" from "how it does it," fostering a clean and modular design.