**Chapter:1**  **Access Modifiers (Specifiers) in C#**

Access modifiers (or access specifiers) in C# define the accessibility level of classes, methods, variables, and other members within a program. They control where a member can be accessed from, providing different levels of encapsulation and security.

C# provides the following access modifiers:

1. **Public (public)**
2. **Private (private)**
3. **Protected (protected)**
4. **Internal (internal)**
5. **Protected Internal (protected internal)**
6. **Private Protected (private protected)**

Let's discuss each of them with examples:

**1. Public (public)**

* **Description**: Members declared with the public modifier are accessible from anywhere within the application, including different classes, assemblies, and projects.
* **Example**:

***public class Car***

***{***

***public string Model;***

***public void StartEngine()***

***{***

***Console.WriteLine("Engine started.");***

***}***

***}***

***class Program***

***{***

***static void Main()***

***{***

***Car myCar = new Car();***

***myCar.Model = "Tesla"; // Accessible***

***myCar.StartEngine(); // Accessible***

***}***

***}***

**2. Private (private)**

* **Description**: Members declared with the private modifier are accessible only within the class or struct in which they are declared. This is the most restrictive access level.
* **Example**:

***public class Car***

***{***

***private string model;***

***private void StartEngine()***

***{***

***Console.WriteLine("Engine started.");***

***}***

***public void SetModel(string modelName)***

***{***

***model = modelName;***

***}***

***}***

***class Program***

***{***

***static void Main()***

***{***

***Car myCar = new Car();***

***// myCar.model = "Tesla"; // Not accessible***

***// myCar.StartEngine(); // Not accessible***

***myCar.SetModel("Tesla"); // Accessible via public method***

***}***

***}***

**3. Protected (protected)**

* **Description**: Members declared with the protected modifier are accessible within the class in which they are declared and in any derived (child) classes. This allows child classes to access the base class's protected members.
* **Example**:

***public class Car***

***{***

***protected string model;***

***protected void StartEngine()***

***{***

***Console.WriteLine("Engine started.");***

***}***

***}***

***public class ElectricCar : Car***

***{***

***public void SetModel(string modelName)***

***{***

***model = modelName; // Accessible in derived class***

***StartEngine(); // Accessible in derived class***

***}***

***}***

***class Program***

***{***

***static void Main()***

***{***

***ElectricCar myCar = new ElectricCar();***

***myCar.SetModel("Tesla"); // Accessible via public method***

***// myCar.model = "Tesla"; // Not accessible***

***// myCar.StartEngine(); // Not accessible***

***}***

***}***

**4. Internal (internal)**

* **Description**: Members declared with the internal modifier are accessible only within the same assembly. They cannot be accessed from another assembly.
* **Example**:

***internal class Car***

***{***

***internal string Model;***

***internal void StartEngine()***

***{***

***Console.WriteLine("Engine started.");***

***}***

***}***

***class Program***

***{***

***static void Main()***

***{***

***Car myCar = new Car();***

***myCar.Model = "Tesla"; // Accessible within the same assembly***

***myCar.StartEngine(); // Accessible within the same assembly***

***}***

***}***

**5. Protected Internal (protected internal)**

* **Description**: Members declared with the protected internal modifier are accessible from the current assembly and from derived classes in other assemblies.
* **Example**:

***public class Car***

***{***

***protected internal string Model;***

***protected internal void StartEngine()***

***{***

***Console.WriteLine("Engine started.");***

***}***

***}***

***public class ElectricCar : Car***

***{***

***public void SetModel(string modelName)***

***{***

***Model = modelName; // Accessible in derived class, even from another assembly***

***StartEngine(); // Accessible in derived class, even from another assembly***

***}***

***}***

***class Program***

***{***

***static void Main()***

***{***

***ElectricCar myCar = new ElectricCar();***

***myCar.SetModel("Tesla"); // Accessible via public method***

***}***

***}***

**6. Private Protected (private protected)**

* **Description**: Members declared with the private protected modifier are accessible within the class in which they are declared and within derived classes that are in the same assembly. This is more restrictive than protected internal.
* **Example**:

***public class Car***

***{***

***private protected string Model;***

***private protected void StartEngine()***

***{***

***Console.WriteLine("Engine started.");***

***}***

***}***

***public class ElectricCar : Car***

***{***

***public void SetModel(string modelName)***

***{***

***Model = modelName; // Accessible in derived class, but only within the same assembly***

***StartEngine(); // Accessible in derived class, but only within the same assembly***

***}***

***}***

***class Program***

***{***

***static void Main()***

***{***

***ElectricCar myCar = new ElectricCar();***

***myCar.SetModel("Tesla"); // Accessible via public method***

***}***

***}***

### Chapter: 2. Encapsulation in C#

#### ****Introduction to Encapsulation****

Encapsulation is one of the fundamental principles of Object-Oriented Programming (OOP) in C#. It involves the bundling of data (fields) and methods (functions) that operate on the data into a single unit, known as a class. The primary goal of encapsulation is to prevent external code from directly accessing and modifying the internal state of an object, thus safeguarding the integrity of the data.

In simpler terms, encapsulation allows you to hide the internal implementation details of a class from the outside world while exposing only the necessary parts. This promotes data protection and enhances code maintainability.

#### ****Real-World Example of Encapsulation****

In the real world, encapsulation can be compared to a capsule that contains medicine. The capsule shields the medicine (data) from the external environment, and one can only access the medicine by opening the capsule in a controlled manner. Similarly, in C#, a class encapsulates its fields, and the outside world interacts with these fields through controlled access points known as properties.

#### ****Encapsulation in C#****

In C#, encapsulation is implemented using access modifiers and properties. By defining class fields with the private access modifier, you can restrict direct access to these fields from outside the class. However, you can still expose these fields through public properties, allowing controlled access to the fields.

Here’s a simple example demonstrating encapsulation in C#:

***class User***

***{***

***private string location;***

***private string name;***

***public string Location***

***{***

***get***

***{***

***return location;***

***}***

***set***

***{***

***location = value;***

***}***

***}***

***public string Name***

***{***

***get***

***{***

***return name;***

***}***

***set***

***{***

***name = value;***

***}***

***}***

***}***

In this example, the fields location and name are private, meaning they cannot be accessed directly from outside the User class. Instead, the class provides public properties Location and Name with get and set accessors that control how these fields can be read and modified.

#### ****Why Use Encapsulation?****

Encapsulation is used to achieve several objectives:

1. **Data Protection:** Encapsulation prevents unauthorized access and accidental modification of an object’s internal state.
2. **Code Maintainability:** By hiding the internal implementation details, you can change the internal workings of a class without affecting external code that depends on it.
3. **Controlled Access:** Through properties, you can control how the internal data is accessed or modified, enabling validation and other logic within the get or set accessors.

#### ****Encapsulation in Action****

Let’s see an example of encapsulation in action within a C# program:

***using System;***

***namespace EncapsulationExample***

***{***

***class User***

***{***

***private string location;***

***private string name;***

***public string Location***

***{***

***get***

***{***

***return location;***

***}***

***set***

***{***

***location = value;***

***}***

***}***

***public string Name***

***{***

***get***

***{***

***return name;***

***}***

***set***

***{***

***name = value;***

***}***

***}***

***}***

***class Program***

***{***

***static void Main(string[] args)***

***{***

***User u = new User();***

***// Using the set accessor***

***u.Name = "Suresh Dasari";***

***u.Location = "Hyderabad";***

***// Using the get accessor***

***Console.WriteLine("Name: " + u.Name);***

***Console.WriteLine("Location: " + u.Location);***

***Console.WriteLine("\nPress Enter Key to Exit..");***

***Console.ReadLine();***

***}***

***}***

***}***

**Explanation:**

* The User class encapsulates the name and location fields.
* These fields are accessed and modified through the Name and Location properties, which provide controlled access to the data.
* In the Main method, we create an instance of the User class and use the properties to set and retrieve the values of name and location.

--------------------------------------------------------------------------------------------------------------------------

### Chapter: Abstraction in C#

#### ****Introduction to Abstraction****

Abstraction is a key principle of Object-Oriented Programming (OOP) that focuses on hiding the complex implementation details of an object and exposing only the essential features that are necessary for the user to interact with the object. By simplifying the interface and concealing the underlying complexity, abstraction helps to manage complexity in software development and enhances code readability and maintainability.

In C#, abstraction is typically achieved using access modifiers that control the visibility of class members (fields, properties, methods). By exposing only the necessary parts of a class to the outside world, you can prevent misuse and ensure that the internal workings of the class are not tampered with.

#### ****Real-World Example of Abstraction****

A laptop is a real-world example of abstraction. A laptop consists of various components such as the processor, RAM, motherboard, screen, keyboard, etc. As a user, you only need to know how to use the laptop by pressing the power button, typing on the keyboard, and viewing the screen. You don’t need to understand how each internal component works or interacts with the others. The laptop abstracts away the complexity, allowing you to interact with it through a simple, user-friendly interface.

Similarly, in C#, classes abstract the internal details and expose only the necessary functionality.

#### ****Abstraction in C#****

In C#, abstraction is implemented through the use of classes and access modifiers. You can define a class with various methods and properties and decide which ones should be exposed to the outside world and which should remain hidden. Public members are accessible from outside the class, while private members are hidden and can only be accessed within the class itself.

Here's an example demonstrating abstraction in C#:

***using System;***

***namespace AbstractionExample***

***{***

***public class Laptop***

***{***

***private string brand;***

***private string model;***

***public string Brand***

***{***

***get { return brand; }***

***set { brand = value; }***

***}***

***public string Model***

***{***

***get { return model; }***

***set { model = value; }***

***}***

***public void LaptopDetails()***

***{***

***Console.WriteLine("Brand: " + Brand);***

***Console.WriteLine("Model: " + Model);***

***}***

***public void LaptopKeyboard()***

***{***

***Console.WriteLine("Typing on the Keyboard");***

***}***

***private void MotherBoardInfo()***

***{***

***Console.WriteLine("MotherBoard Information");***

***}***

***private void InternalProcessor()***

***{***

***Console.WriteLine("Processor Information");***

***}***

***}***

***class Program***

***{***

***static void Main(string[] args)***

***{***

***Laptop laptop = new Laptop();***

***laptop.Brand = "Dell";***

***laptop.Model = "Inspiron 14R";***

***laptop.LaptopDetails();***

***Console.WriteLine("\nPress Enter Key to Exit..");***

***Console.ReadLine();***

***}***

***}***

***}***

In this example:

* The Laptop class encapsulates several fields (brand and model), properties (Brand and Model), and methods (LaptopDetails, LaptopKeyboard, MotherBoardInfo, and InternalProcessor).
* The Brand and Model properties, along with the LaptopDetails and LaptopKeyboard methods, are public and can be accessed from outside the class.
* The MotherBoardInfo and InternalProcessor methods are private, meaning they cannot be accessed or called from outside the Laptop class.

By using abstraction, the Laptop class exposes only the necessary functionality (Brand, Model, LaptopDetails, and LaptopKeyboard) to the outside world, while hiding the internal workings of the motherboard and processor.

#### ****Difference Between Abstraction and Encapsulation****

While both abstraction and encapsulation are fundamental principles of OOP, they serve different purposes:

| **Abstraction** | **Encapsulation** |
| --- | --- |
| Hides unnecessary details and shows only the required properties and methods. | Binds data members and member functions into a single unit to prevent outsiders from accessing it directly. |
| Achieved through classes and access modifiers that hide implementation details. | Achieved by defining classes with private fields and public methods/properties to control access. |
| Focuses on what an object does. | Focuses on how an object is implemented. |

#### ****Conclusion****

Abstraction is a powerful concept in C# that allows developers to simplify complex systems by hiding unnecessary details and exposing only the essential features. By using access modifiers and careful class design, you can achieve abstraction, making your code easier to understand, maintain, and extend.

--------------------------------------------------------------------------------------------------------------------------