Tab 1

# **C# Programming – Lecture 4**

## **1. Properties in C#**

### **1.1 Why Properties Are Needed**

In early programming languages, class data members were often made public, allowing direct access:

class Student

{

public int age;

}

This approach is **unsafe** because:

* No validation (negative age allowed)
* No control over data modification
* Breaks **Encapsulation**
* Internal data can be misused from outside the class

**Encapsulation Rule**

Data should be hidden and accessed only through controlled mechanisms. C# solves this problem using **Properties**, which act as **controlled gateways** to private fields.



### **1.2 What Is a Property?**

A **Property** is a class member that:

* Exposes a private field safely
* Controls **read (get)** and **write (set)** access
* Can include **business rules / validation logic**
* Looks like a variable but behaves like a method

**General Syntax**

access\_modifier data\_type PropertyName

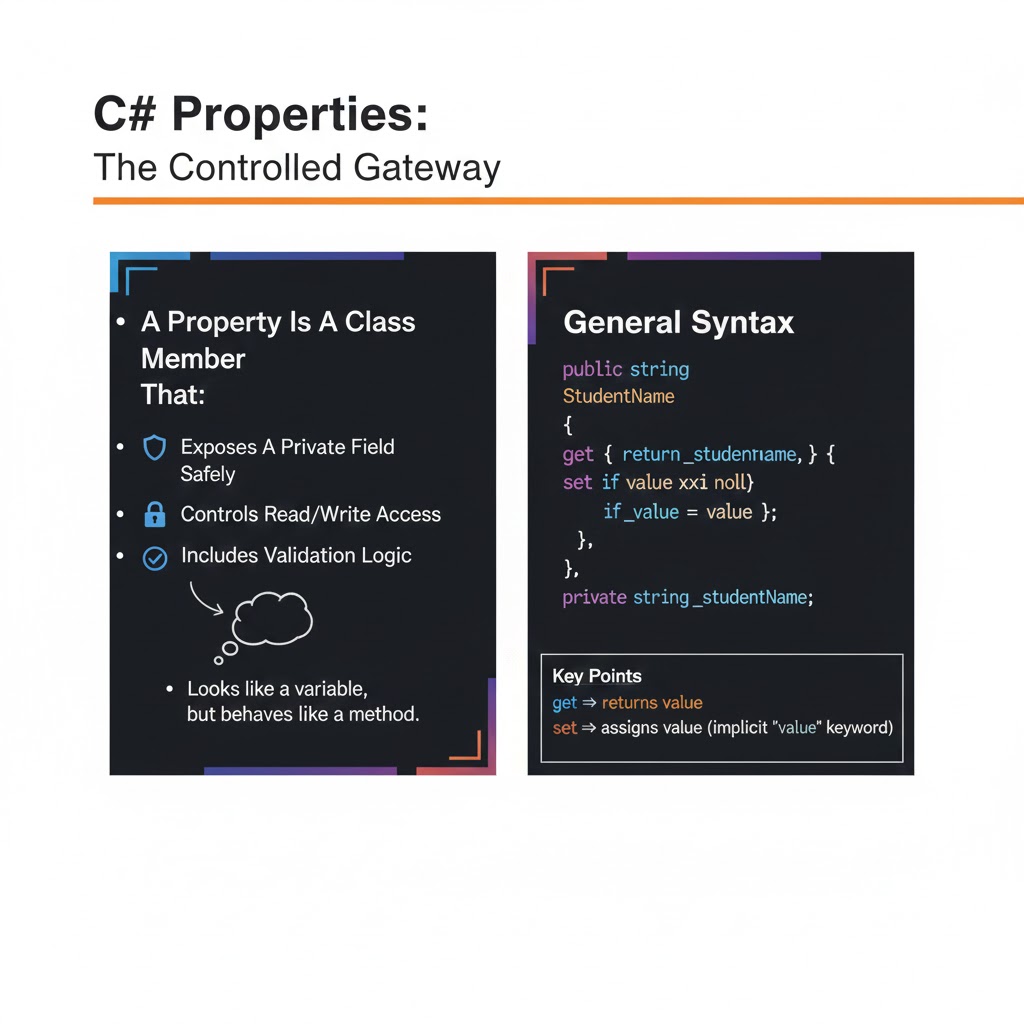
{

get { return field; }

set { field = value; }

}

**Key Point**

* get → returns value
* set → assigns value (value is an implicit keyword)  
  

# **Core Question**

**Why do we need Properties when we already have private variables with public methods?**

Short answer first:

**Properties are needed because they give controlled access to data in a simpler, safer, and more standard way than public methods.**

Now let’s really understand *why*.

## **Step 1: What you already know (using methods)**

You already understand this idea:

* Data is **private**
* Methods are **public**
* Methods control access

This is correct.

So yes — **methods CAN do the job**.

## **Step 2: Then what problem still exists?**

The problem is **how the outside world interacts with data**.

Think like a user of a class.

### **With public methods, access looks like this:**

* GetBalance()
* SetBalance(5000)

### **With properties, access looks like this:**

* Balance
* Balance = 5000

## **Step 3: Why does this difference matter?**

Because **data access and behavior are not the same thing**.

### **Real-life analogy (very important)**

#### **ATM Machine**

* Checking balance → **data access**
* Withdrawing money → **behavior**

You don’t think of “checking balance” as an *action*.  
 You think of it as *seeing a value*.

Properties model this thinking correctly.

## **Step 4: Conceptual difference (very important)**

| **Concept** | **Methods** | **Properties** |
| --- | --- | --- |
| Purpose | Perform actions | Expose data |
| How it feels | Verb (Do something) | Noun (Value) |
| Example | CalculateSalary() | Salary |
| Usage style | Function call | Variable access |

## **Step 5: Why methods are NOT ideal for data access**

### **Problem 1: Methods break readability**

Compare:

* employee.GetAge()
* employee.Age

Which one feels like **data**?

Clearly → Age

### **Problem 2: Methods allow wrong design thinking**

With methods, developers may:

* Add logic later
* Change behavior
* Mix actions with data

This creates **confusion** in large systems.

Properties clearly say:

“This is DATA, not an ACTION.”

### **Problem 3: Frameworks & tools expect properties**

This is a **very big reason**.

Modern systems like:

* Data binding
* Serialization
* ORMs (Entity Framework)
* UI frameworks

**DO NOT work well with Get/Set methods**.

They expect:

* obj.Name
* obj.Age

Not:

* obj.GetName()

So properties are **industry-standard**, not optional.

## **Step 6: But aren’t properties just methods inside?**

YES — and that’s the key idea.

**A property IS a method in disguise.**

Internally:

* get → method
* set → method

Externally:

* Looks like a variable

This gives the **best of both worlds**:

* Safety of methods
* Simplicity of variables

### 

### **1.3 Read/Write Properties (Full Properties)**

class Employee

{

private double salary;

public double Salary

{

get { return salary; }

set

{

if (value > 0)

salary = value;

}

}

}

* Ensures salary is always positive
* Prevents invalid data
* Demonstrates **Encapsulation + Validation**

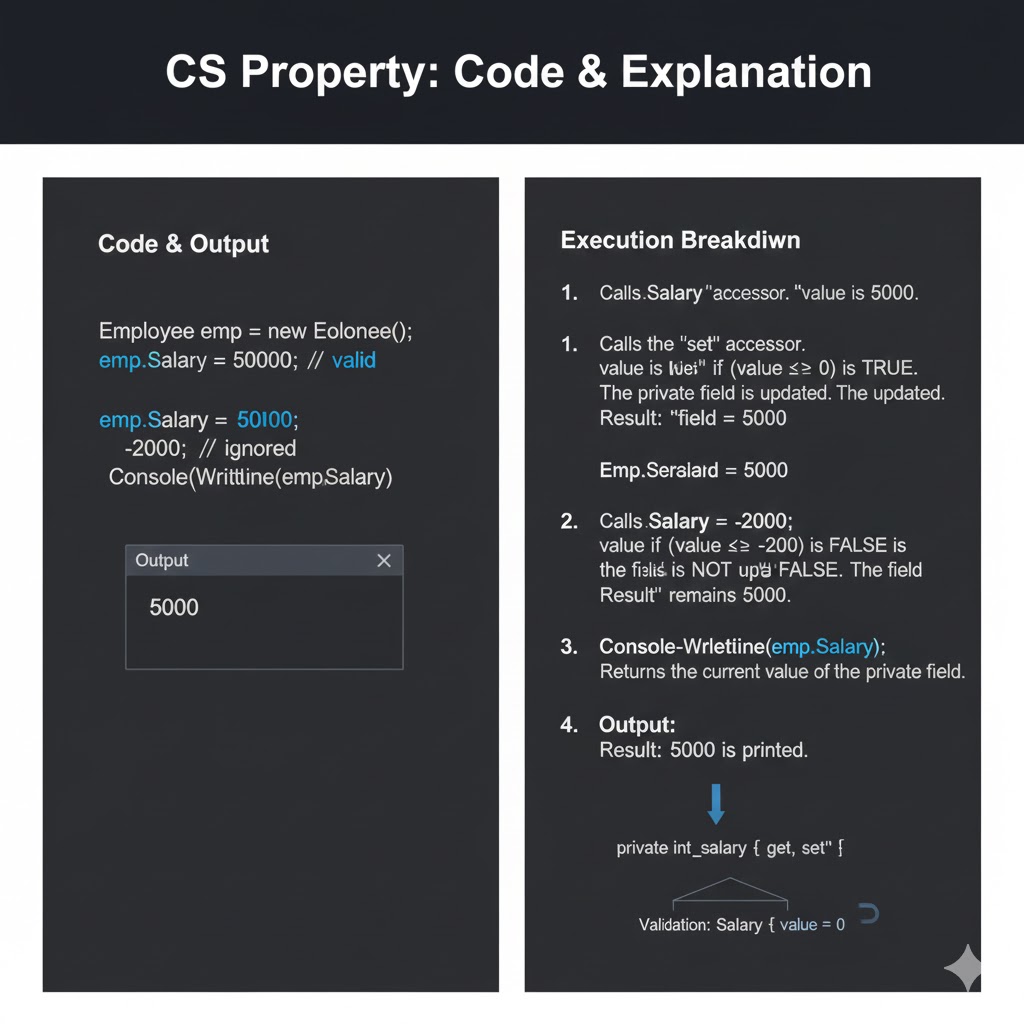
**Usage**

Employee emp = new Employee();

emp.Salary = 50000; // valid

emp.Salary = -2000; // ignored

Console.WriteLine(emp.Salary);



### **1.4 Auto-Implemented Properties**

When no extra logic is required, C# allows **auto-implemented properties**.

class Product

{

public int ProductId { get; set; }

public string ProductName { get; set; }

public double Price { get; set; }

}

* Compiler creates the backing field automatically
* Less code, more readability
* Used heavily in **DTOs, Models, APIs**

**Equivalent to**

private int productId;

public int ProductId

{

get { return productId; }

set { productId = value; }

}

### 

## **Problem Statement**

You are designing a **Student Profile System**.

Each student has:

1. **Name**
2. **Age**
3. **Marks**

### **Rules (Business Logic)**

* Name cannot be empty
* Age must be greater than 0
* Marks must be between 0 and 100

You must:

* Store data using **private variables**
* Expose data using **public properties**
* Apply validation inside set

## **Task Requirements**

### **Step 1: Create a class Student**

The class should contain:

* Private fields:  
  + name
  + age
  + marks

### **Step 2: Create Properties**

Create the following properties:

#### **1. Name Property**

* get → returns name
* set → assigns name only if it is not empty

#### **2. Age Property**

* get → returns age
* set → allows age only if greater than 0

#### **3. Marks Property**

* get → returns marks
* set → allows marks only between 0 and 100

### **Step 3: Use the Properties**

In Main():

* Create a student object
* Assign values using properties

# **ANSWER: Student Profile Management Using Properties (C#)**

## **1. Class Implementation Using Properties**

class Student

{

// Private fields (data hiding)

private string name;

private int age;

private int marks;

// Property for Name

public string Name

{

get

{

return name;

}

set

{

if (!string.IsNullOrEmpty(value))

{

name = value;

}

}

}

// Property for Age

public int Age

{

get

{

return age;

}

set

{

if (value > 0)

{

age = value;

}

}

}

// Property for Marks

public int Marks

{

get

{

return marks;

}

set

{

if (value >= 0 && value <= 100)

{

marks = value;

}

}

}

}

## **2. Using the Properties (Main Logic)**

class Program

{

static void Main()

{

Student s = new Student();

s.Name = "Amit";

s.Age = 20;

s.Marks = 85;

Console.WriteLine("Name: " + s.Name);

Console.WriteLine("Age: " + s.Age);

Console.WriteLine("Marks: " + s.Marks);

}

}

## **3. Explanation of the Code (Important for Exams)**

### **Why fields are private**

* Prevents direct access to data
* Ensures data security
* Forces controlled access through properties

### **Why properties are public**

* Allow safe access to private data
* Apply validation rules
* Maintain encapsulation

### **Role of get**

* Returns the value of the private field
* Used when data is read

### **Role of set**

* Assigns a value to the private field
* Executes validation logic before assignment
* Uses the implicit keyword value

## **4. Conceptual Questions – Answered**

### **1. Why can’t we make variables public directly?**

Public variables allow anyone to modify data without validation, which can lead to invalid or unsafe data.

### **2. What happens if invalid data is assigned through a property?**

The validation logic inside set prevents the assignment, so the data remains unchanged.

### **3. Where does the value keyword get its data from?**

value receives the data from the right-hand side of the assignment statement.

Example:

s.Age = 20;

Here, value = 20.

### **4. Can a property have only get or only set?**

Yes.

* Only get → Read-only property
* Only set → Write-only property

### **5. Why do properties look like variables but act like methods?**

They are accessed like variables, but internally execute code through get and set methods.

### **1.5 Read-Only Properties**

Used when a value should **not change after initialization**.

class Circle

{

private double radius;

public Circle(double r)

{

radius = r;

}

public double Area

{

get { return Math.PI \* radius \* radius; }

}

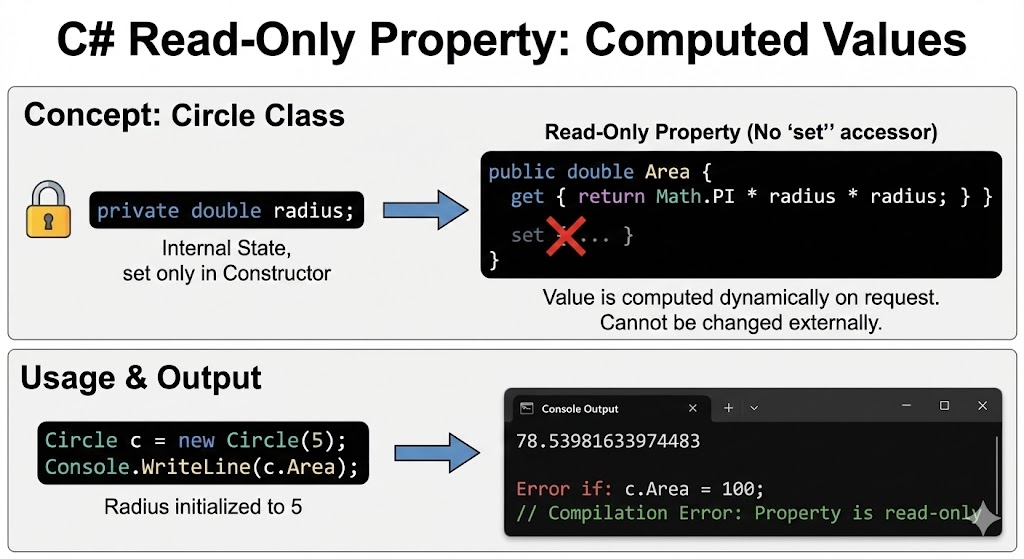
}

* No set block
* Value is computed dynamically
* Common for **derived/calculated value**

**Example Usage**

Circle c = new Circle(5);

Console.WriteLine(c.Area);



### **1.6 Write-Only Properties**

Used when data should be **accepted but never revealed**.

class User

{

private string password;

public string Password

{

set { password = value; }

}

}

* Protects sensitive data
* Used for passwords, secrets, tokens
* No get → data cannot be read

**Note:** Rare in real-world apps; usually hashing is preferred.

## 

# **EXTENDED TASK: Advanced Properties in C#**

## **Task Title**

**Student Profile System – Advanced Property Types**

## **Objective**

To understand and implement:

1. Normal properties with validation
2. Auto-implemented properties
3. Read-only properties
4. Write-only properties

## **Base Scenario**

You are developing a **Student Profile System** for a college application.

Each student has:

* Name
* Age
* Marks
* Student ID
* Password

Different data requires **different access rules**.

## **PART A: Auto-Implemented Property**

### **Requirement**

Store the **Student ID**.

Rules:

* No validation required
* Direct get and set allowed

### **Task Instruction**

Use an **auto-implemented property**.

### **Hint**

Auto-implemented properties do not require a private backing field.

## **PART B: Read-Only Property**

### **Requirement**

Calculate **Result Status** based on marks.

Rules:

* Marks ≥ 40 → “Pass”
* Marks < 40 → “Fail”
* Value should be readable
* Value should NOT be set from outside

### **Task Instruction**

Create a **read-only property**.

## **PART C: Write-Only Property**

### **Requirement**

Store **Password** securely.

Rules:

* Password can be set
* Password must NOT be readable
* Password length must be at least 6 characters

### **Task Instruction**

Create a **write-only property**.

## **PART D: Normal Property with Validation (Revision)**

### **Requirement**

* Name cannot be empty
* Age must be greater than 0
* Marks must be between 0 and 100

class Student

{

// Normal private fields

private string name;

private int age;

private int marks;

private string password;

// PART A: Auto-Implemented Property

public int StudentId { get; set; }

// PART D: Normal Property - Name

public string Name

{

get

{

return name;

}

set

{

if (!string.IsNullOrEmpty(value))

{

name = value;

}

}

}

// PART D: Normal Property - Age

public int Age

{

get

{

return age;

}

set

{

if (value > 0)

{

age = value;

}

}

}

// PART D: Normal Property - Marks

public int Marks

{

get

{

return marks;

}

set

{

if (value >= 0 && value <= 100)

{

marks = value;

}

}

}

// PART B: Read-Only Property

public string Result

{

get

{

return marks >= 40 ? "Pass" : "Fail";

}

}

// PART C: Write-Only Property

public string Password

{

set

{

if (value.Length >= 6)

{

password = value;

}

}

}

}

## **1.7 Properties with Private Set**

class Account

{

public int AccountNo { get; private set; }

public Account(int accNo)

{

AccountNo = accNo;

}

}

## **What Does “Read-Only from Outside, Writable Only Inside the Class” Mean?**

It means:

* **Any code outside the class can only read the value**
* **Only the class itself can modify the value**

This is achieved using a **public get and a private set**.

## **Line-by-Line Explanation**

### **class Account**

* Defines a class named Account
* Represents a bank account entity

### **public int AccountNo { get; private set; }**

This is the **key line**. Let’s break it clearly.

#### **public**

* The property is accessible from outside the class

#### **int**

* Data type of the account number

#### **AccountNo**

* Property name

#### **get**

* Public by default
* Allows **reading the account number** from outside the class

#### **private set**

* The set accessor is **private**
* Only code inside the Account class can modify the value
* External code **cannot change it**

This means:

* Console.WriteLine(account.AccountNo); // Allowed
* account.AccountNo = 123; // Not allowed

### **public Account(int accNo)**

* Constructor of the class
* Runs when an object is created
* Receives the account number as input

### **AccountNo = accNo;**

* Assignment is allowed because:
  + Constructor is **inside the same class**
  + private set is accessible here
* Sets the account number **once at creation time**

## **How the Class Is Used (Conceptually)**

* Account acc = new Account(1001);
* Console.WriteLine(acc.AccountNo); // Allowed
* acc.AccountNo = 2002; // Compile-time error

## **Why This Design Is Important**

### **1. Prevents Accidental Modification**

Once an account number is assigned:

* It should never change
* External code cannot modify it by mistake

### **2. Ensures Data Integrity**

Account numbers are:

* Unique identifiers
* Critical banking data

Allowing free modification could lead to:

* Data corruption
* Transaction mismatches
* Security issues

### **3. Controlled Changes (If Ever Needed)**

If the bank needs to change the account number:

* It can be done **inside the class only**
* Through controlled methods
* With proper validation and logging

## **Why This Is Common in Banking Systems**

In banking:

* Account number is issued once
* Customers, UI, APIs should only **view**, not modify it
* Only the bank’s internal system should control changes

private set perfectly matches this rule.

## **Difference from Read-Only Property**

| **Feature** | **Read-Only Property** | **Private Set Property** |
| --- | --- | --- |
| Has get | Yes | Yes |
| Has set | No | Yes (private) |
| Can change inside class | ❌ No | ✔ Yes |
| Flexibility | Low | Higher |

## **Exam-Ready Definition**

**A property with a private set allows public read access while restricting write access to within the class, ensuring data integrity and controlled modification.**

## **One-Line Memory Rule**

**public get + private set = read everywhere, write only inside**

## **1.8**

## **What Are Init-Only Properties?**

An **init-only property** is a property whose value can be assigned **only at the time of object creation** and **cannot be changed later**.

It is defined using the **init accessor** instead of set.

## **Why Init-Only Properties Were Introduced (C# 9)**

Before C# 9:

* Properties used set
* Any code could change values anytime

Problem:

* Objects could be modified accidentally
* Hard to create **immutable objects**

Solution:

* init allows **one-time assignment**
* Object becomes **read-only after creation**

## **Code Given**

class Student

{

public int RollNo { get; init; }

public string Name { get; init; }

}

## **Line-by-Line Explanation**

### **class Student**

* Declares a class named Student
* Represents a student entity

### **public int RollNo { get; init; }**

* public → accessible from outside
* int → data type
* RollNo → property name
* get → allows reading the value
* init → allows setting the value **only during initialization**

Important:

* After object creation, this property becomes **read-only**

### **public string Name { get; init; }**

* Same logic as RollNo
* Stores student name
* Can be assigned only once

## **Object Creation Using Init-Only Properties**

Student s = new Student

{

RollNo = 1,

Name = "Rahul"

};

### **What happens here?**

* Object is created
* Properties are assigned **during initialization**
* Compiler allows this assignment

This is called **object initializer syntax**.

## **Why This Assignment Is NOT Allowed**

s.RollNo = 2; // Compile-time error

Reason:

* Object creation is already complete
* init does not allow modification after initialization
* Compiler blocks the assignment

## **How Init Is Different from Set**

| **Feature** | **set** | **init** |
| --- | --- | --- |
| Can assign during initialization | Yes | Yes |
| Can modify after creation | Yes | No |
| Supports immutability | No | Yes |
| Introduced in | Early C# | C# 9 |

## **What Is Immutability?**

**Immutable object**:

* Once created, its state **cannot change**

Init-only properties help create such objects easily.

Example:

* Roll number and name should not change once student is registered

## **Real-World Use Cases**

### **1. APIs**

* Data received from request should not be changed
* Ensures data consistency

### **2. Configuration Objects**

* Settings loaded once
* Prevent accidental modification

### **3. Records (C# 9+)**

* Records use init-only properties by default
* Designed for immutable data models

## **Why Init-Only Properties Are Safer**

* Prevent accidental updates
* Easier to debug
* Thread-safe by design
* Cleaner and more predictable code

## **Exam-Ready Definition**

**Init-only properties are C# properties that allow value assignment only during object initialization, enabling the creation of immutable objects.**

## **One-Line Memory Rule**

**init = set once, then freeze**

## 

## **1.9**

## **What Are Expression-Bodied Properties?**

An **expression-bodied property** is a **shorter way to write a read-only property** when the property only **returns a single expression**.

Instead of writing a full get block, we use the **=> (lambda) operator**.

## **Code Given**

class Rectangle

{

public double Length { get; set; }

public double Width { get; set; }

public double Area => Length \* Width;

}

## **Line-by-Line Explanation**

### **class Rectangle**

* Defines a class named Rectangle
* Represents a geometric shape

### **public double Length { get; set; }**

* An **auto-implemented property**
* Stores the length of the rectangle
* Has both get and set
* No validation is added, so simple syntax is used

### **public double Width { get; set; }**

* Another auto-implemented property
* Stores the width of the rectangle
* Also has get and set

### **public double Area => Length \* Width;**

This is the **expression-bodied property**.

Let’s break it down carefully.

#### **public double Area**

* Declares a public property named Area
* The return type is double

#### **=>**

* Called the **expression-bodied operator**
* Means:  
  “Return the result of the expression on the right”

#### **Length \* Width**

* This expression calculates the area of the rectangle
* It uses existing properties
* No additional logic is required

So this line means:

“Whenever Area is accessed, calculate and return Length × Width.”

## **What Happens Internally?**

The compiler treats this:

public double Area => Length \* Width;

as the following traditional code:

public double Area

{

get

{

return Length \* Width;

}

}

So:

* It is **read-only**
* It is **calculated dynamically**
* No value is stored in memory

## **Why Expression-Bodied Properties Are Cleaner**

### **1. Less Code**

No get, no braces, no return keyword.

### **2. Better Readability**

The formula is visible **at a glance**.

### **3. Ideal for Calculated Properties**

Used when:

* Value depends on other properties
* No validation or branching logic is needed

Examples:

* Area
* Percentage
* Total
* Average
* FullName

## **When Should You Use Expression-Bodied Properties?**

Use them when:

* Property is **read-only**
* Property returns **a single expression**
* No if, loop, or complex logic is required

Do **not** use them when:

* Validation is needed
* Multiple statements are required
* Property needs a set

## **Real-World Meaning**

In real applications:

* Area is not stored
* It is always calculated from Length and Width
* Expression-bodied properties ensure **data consistency**

## **Exam-Ready Definition**

**An expression-bodied property is a concise way to define a read-only property in C# using the => operator to return a single expression.**

## **EXTENDED TASK 2 - STUDENT**

## **PART 5: Property with Private Set**

### **Requirement**

Store the **Registration Number**.

### **Rules**

* Value should be readable from outside
* Value should be modifiable **only inside the class**

### **Task Instruction**

* Use a property with **public get and private set**
* Assign the value internally (constructor or method)

## **PART 6: Init-Only Property**

### **Requirement**

Store the **Admission Year**.

### **Rules**

* Value should be set only at the time of object creation
* Value must not be modified later

### **Task Instruction**

* Use an **init-only property**
* Set the value using object initializer syntax

## **PART 7: Expression-Bodied Property**

### **Requirement**

Create a property to calculate **Percentage**.

### **Rules**

* Percentage = (Marks / 100) × 100
* Property should be read-only
* Use shortest possible syntax

### **Task Instruction**

* Implement an **expression-bodied property**

## **PART 8: Usage Task**

In the Main() method:

* Create a Student object
* Assign values using all allowed properties
* Display all readable properties
* Try accessing restricted properties and note compiler errors

# **ANSWER: Student Profile System – Advanced Properties in C#**

## **1. Student Class Implementation**

class Student

{

// =========================

// PART 1: Normal Properties

// =========================

private string name;

private int age;

private int marks;

private string password;

// ==============================

// PART 5: Property with Private Set

// ==============================

public int RegistrationNumber { get; private set; }

// =========================

// PART 6: Init-Only Property

// =========================

public int AdmissionYear { get; init; }

// =========================

// PART 7: Expression-Bodied Property

// =========================

public double Percentage => (marks / 100.0) \* 100;

// Constructor (sets private-set property)

public Student(int regNo)

{

RegistrationNumber = regNo;

}

}

## **2. Using the Student Class (Main Method)**

class Program

{

static void Main()

{

Student s = new Student(5001)

{

AdmissionYear = 2023

};

s.StudentId = 101;

s.Name = "Amit";

s.Age = 20;

s.Marks = 78;

s.Password = "secure123";

Console.WriteLine("Student ID: " + s.StudentId);

Console.WriteLine("Name: " + s.Name);

Console.WriteLine("Age: " + s.Age);

Console.WriteLine("Marks: " + s.Marks);

Console.WriteLine("Result: " + s.Result);

Console.WriteLine("Percentage: " + s.Percentage);

Console.WriteLine("Registration No: " + s.RegistrationNumber);

Console.WriteLine("Admission Year: " + s.AdmissionYear);

}

}

## **3. Invalid Operations (Compile-Time Errors)**

s.Result = "Pass"; // Error: Read-only

Console.WriteLine(s.Password); // Error: Write-only

s.RegistrationNumber = 6001; // Error: private set

s.AdmissionYear = 2024; // Error: init-only

## **4. Conceptual Answers (Exam-Ready)**

### **1. Why fields are private and properties public**

Private fields protect data, while public properties provide controlled and validated access.

### **2. Auto-implemented property use**

Used when no validation or logic is required; reduces boilerplate code.

### **3. Read-only property use**

Used for calculated or derived values that should not be modified externally.

### **4. Write-only property use**

Used to protect sensitive data like passwords from being read.

### **5. Difference: private set vs init**

* private set → value can change inside class anytime
* init → value can be set only during object creation

### **6. Why expression-bodied properties are preferred**

They improve readability and reduce unnecessary code for simple calculations.

## **5. Summary Table (High-Scoring)**

| **Property Type** | **Get** | **Set** | **Purpose** |
| --- | --- | --- | --- |
| Normal | Yes | Yes | Validation |
| Auto | Yes | Yes | Simple data |
| Read-only | Yes | No | Derived values |
| Write-only | No | Yes | Sensitive data |
| Private set | Yes | Restricted | IDs |
| Init-only | Yes | Once | Immutable data |
| Expression-bodied | Yes | No | Calculations |

## **Final Exam Conclusion**

**C# provides multiple property types to enforce access control, validation, immutability, and clean design. Choosing the correct property type ensures data safety, maintainability, and real-world business rule enforcement.**

## **1.10 Properties vs Fields**

| **Feature** | **Field** | **Property** |
| --- | --- | --- |
| Validation | No | Yes |
| Encapsulation | No | Yes |
| Logic Allowed | No | Yes |
| Recommended | No | Yes |

## 

## 

## 

## 

## 

## 

## **2. Indexers in C#**

## **2.1 What Is an Indexer?**

An **Indexer** allows objects of a class to be accessed **like arrays** using square brackets.

object[index]

An indexer:

* Enables array-like access to class objects
* Is useful when a class represents a **collection of values**
* Improves readability and usability of custom data structures

Instead of calling methods like:

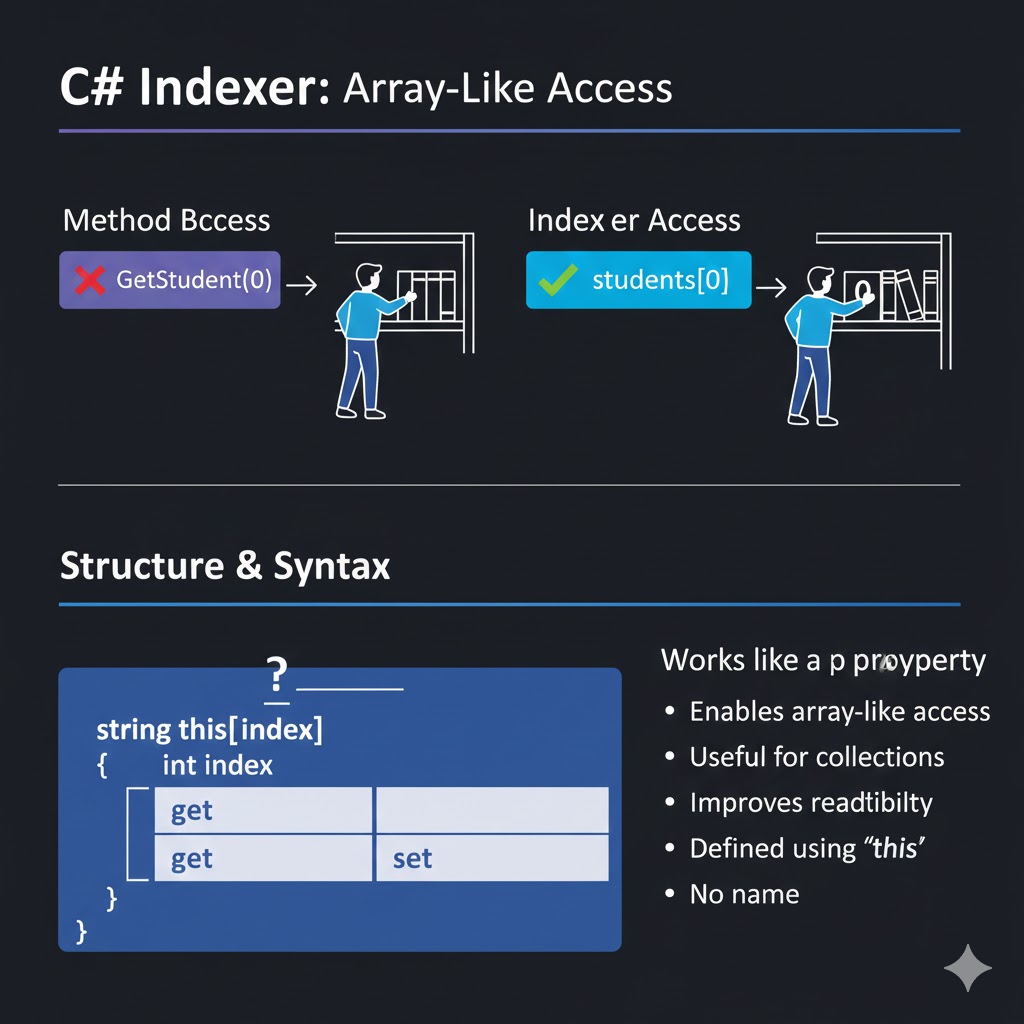
GetStudent(0);

You can write:

students[0];

### **Key Characteristics**

* Defined using the this keyword
* Does not have a name
* Works like a property but takes parameters
* Can have get and set accessors



## **2.2 Why Indexers Are Needed**

Without indexers:

class StudentCollection

{

private string[] students = new string[3];

public string GetStudent(int index)

{

return students[index];

}

public void SetStudent(int index, string name)

{

students[index] = name;

}

}

With indexers:

students[0] = "Amit";

Console.WriteLine(students[0]);

### **Benefits**

* Cleaner syntax
* Easier to use
* Looks and feels like built-in collections
* Improves code readability

## **2.3 Creating an Indexer**

class StudentCollection

{

private string[] students = new string[3];

public string this[int index]

{

get { return students[index]; }

set { students[index] = value; }

}

}

### **Usage**

StudentCollection sc = new StudentCollection();

sc[0] = "Amit";

sc[1] = "Neha";

sc[2] = "Rahul";

Console.WriteLine(sc[0]);

### **Output**

Amit

## **Object as an Array:** By using the this keyword, you allow the sc object to behave like an array, enabling the use of square brackets [] to access internal data.

## **Direct Mapping:** The indexer acts as a bridge; when you write sc[0], the code automatically routes that request to the private students array at position 0.

## **Syntactic Sugar:** It replaces the need for bulky methods like GetStudent() or SetStudent(), making your custom classes much cleaner and more intuitive to use.

## **2.4 Indexer with Validation**

Indexers can include validation logic just like properties.

class StudentCollection

{

private string[] students = new string[3];

public string this[int index]

{

get

{

if (index < 0 || index >= students.Length)

return "Invalid Index";

return students[index];

}

set

{

if (index >= 0 && index < students.Length)

students[index] = value;

}

}

}

This prevents:

* Index out of range errors
* Invalid memory access

### **Example Usage & Output**

StudentCollection sc = new StudentCollection();

sc[0] = "Amit"; // Valid: index 0

sc[5] = "Zeeshan"; // Invalid: index 5 (Ignored by set)

Console.WriteLine(sc[0]); // Output: Amit

Console.WriteLine(sc[5]); // Output: Invalid Index

## **2.5 Read-Only Indexer**

An indexer can be read-only by removing the set accessor.

class Marks

{

private int[] marks = { 80, 90, 85 };

public int this[int index]

{

get { return marks[index]; }

}

}

Use case:

* When data should not be modified from outside
* Reporting and analytics systems

## **2.6 Write-Only Indexer**

Less common but possible.

class SecureData

{

private string[] data = new string[3];

public string this[int index]

{

set { data[index] = value; }

}

}

Use case:

* Storing sensitive information
* Logging systems

**Correct Usage vs. Incorrect Usage**

SecureData sd = new SecureData();

sd[0] = "TopSecret123"; // Works (Data is stored)

string s = sd[0]; // Error (Cannot read)

Console.WriteLine(sd[0]); // Error (Cannot read)

## **2.7 Indexer Overloading**

Indexers can be overloaded using **different parameter types**.

class EmployeeDirectory

{

private Dictionary<int, string> employees = new Dictionary<int, string>();

public string this[int id]

{

get { return employees[id]; }

set { employees[id] = value; }

}

public string this[string name]

{

get

{

return employees.FirstOrDefault(e => e.Value == name).Value;

}

}

}

### **Usage**

EmployeeDirectory ed = new EmployeeDirectory();

ed[101] = "Ravi";

Console.WriteLine(ed[101]);

Console.WriteLine(ed["Ravi"]);

### **Output**

### Ravi

### 101

# **Line-by-Line Explanation**

## **class EmployeeDirectory**

* Defines a class named EmployeeDirectory
* Represents a **collection of employees**
* Designed to behave like a searchable directory

## **private Dictionary<int, string> employees**

private Dictionary<int, string> employees = new Dictionary<int, string>();

### **Explanation**

* Declares a **private dictionary**
* int → employee ID (key)
* string → employee name (value)
* private ensures:  
  + Direct access is restricted
  + All access happens through indexers
* This maintains **encapsulation**

# **First Indexer: Access by Employee ID**

public string this[int id]

{

get { return employees[id]; }

set { employees[id] = value; }

}

### **What this indexer does**

* Allows accessing employees **by ID**
* Enables array-like syntax:

directory[101] = "Rahul";

Console.WriteLine(directory[101]);

### **Breakdown**

#### **public**

* Accessible from outside the class

#### **string**

* Return type of the indexer

#### **this[int id]**

* Defines an indexer that accepts an **integer**
* id represents the employee ID

### **get**

get { return employees[id]; }

* Runs when reading:

string name = directory[101];

* Fetches the name associated with the given ID
* Throws an exception if the key does not exist

### **set**

set { employees[id] = value; }

* Runs when assigning:

directory[101] = "Rahul";

* value contains "Rahul"
* Adds or updates the dictionary entry

## **Purpose of This Indexer**

* Acts like a **dictionary wrapper**
* Simplifies access
* Hides internal storage logic

# **Second Indexer: Access by Employee Name**

public string this[string name]

{

get

{

return employees.FirstOrDefault(e => e.Value == name).Value;

}

}

## **What this indexer does**

* Allows searching employees **by name**
* Enables syntax like:

string result = directory["Rahul"];

## **Breakdown**

#### **this[string name]**

* Defines another indexer
* Accepts a **string parameter**
* This is called **indexer overloading**

C# allows multiple indexers as long as:

* Parameter types are different

### **get logic explained**

employees.FirstOrDefault(e => e.Value == name).Value;

Step-by-step:

1. employees → dictionary
2. .FirstOrDefault(...) → LINQ method
3. e.Value == name → checks employee name
4. Returns the **first matching entry**
5. .Value extracts the employee name

### **Important Note (Conceptual)**

* This indexer:  
  + Searches **values**, not keys
  + Returns the matching name
* If no match is found:  
  + .Value returns null

## **Why This Indexer Has No set**

Because:

* Searching by name is **read-only**
* Writing by name could cause ambiguity
* IDs should control updates, not names

### **Advantage**

* Access data using different keys
* Flexible and intuitive design

## **2.8 Indexers vs Properties**

| **Feature** | **Property** | **Indexer** |
| --- | --- | --- |
| Name | Has name | Uses this |
| Parameters | Not allowed | Allowed |
| Access Syntax | obj.Property | obj[index] |
| Use Case | Single value | Collection of values |

## **2.9 Rules for Indexers**

* Defined using this keyword
* Can be overloaded
* Can have access modifiers
* Can be abstract or virtual
* Can be implemented in interfaces
* Cannot be static

## **2.10 Real-World Use Cases**

* **API Wrappers:** Used to simplify access to JSON responses, allowing you to write response["status"] instead of navigating complex object trees.
* **Custom Caching:** Implementing a cache where you can retrieve stored values using a unique key, such as myCache["user\_101"].
* **Game Development:** Managing a game grid or map where you can access coordinates using map[x, y].
* **File Systems:** Representing a folder structure where files can be accessed by their filename string through an indexer.

# **TASK: Implement Indexer Overloading in C#**

## **Objective**

To understand and implement **indexer overloading** by allowing a class to be accessed using **different parameter types**.

## **Scenario**

You are designing a **Library Management System**.  
Each book in the library has:

* A **Book ID** (integer)
* A **Book Title** (string)

Users should be able to:

* Access books using **Book ID**
* Access books using **Book Title**

This must be achieved using **indexer overloading**, not methods.

## **Task Requirements**

You must create a class named **Library** and implement the following.

## **PART 1: Internal Data Storage**

* Use a suitable collection to store book data
* Book ID should be the key
* Book Title should be the value
* Data structure must be **private**

## **PART 2: Integer-Based Indexer**

### **Requirement**

* Allow access to book title using **Book ID**
* Support both reading and writing

### **Example Usage**

library[101] = "C# Basics";

Console.WriteLine(library[101]);

## **PART 3: String-Based Indexer**

### **Requirement**

* Allow access using **Book Title**
* Return the first matching title
* Read-only access only

### **Example Usage**

Console.WriteLine(library["C# Basics"]);

## **PART 4: Constraints**

* Do not expose the internal collection directly
* Do not use GetBook() or SetBook() methods
* Use indexers only

## **PART 5: Usage Task**

In Main():

* Create a Library object
* Add at least 3 books using integer indexer
* Retrieve books using both indexers
* Observe behavior when a title is not found

## **Conceptual Questions (Write Answers)**

1. What is indexer overloading?
2. How does C# differentiate between overloaded indexers?
3. Why is the string-based indexer read-only?
4. Difference between indexer overloading and method overloading
5. Why indexers improve usability over methods

## **Submission Rules**

* Follow proper naming conventions
* Write clean and readable code
* Add comments explaining indexer behavior
* Handle null or missing values gracefully

## **Learning Outcome**

After completing this task, you should clearly understand:

* Multiple indexers in one class
* How parameter types control indexer selection
* Real-world use of indexer overloading

### **Memory Rule**

**Same object, different index types → indexer overloading**

# **ANSWER: Indexer Overloading in C#**

## **1. Class Implementation: Library**

using System;

using System.Collections.Generic;

using System.Linq;

class Library

{

// Internal data storage (encapsulation)

private Dictionary<int, string> books = new Dictionary<int, string>();

// =========================

// Integer-based Indexer

// =========================

// Access books using Book ID

public string this[int bookId]

{

get

{

return books.ContainsKey(bookId) ? books[bookId] : "Book not found";

}

set

{

books[bookId] = value;

}

}

// =========================

// String-based Indexer

// =========================

// Access books using Book Title (read-only)

public string this[string title]

{

get

{

return books.FirstOrDefault(b => b.Value == title).Value;

}

}

}

## **2. Using the Overloaded Indexers (Main Method)**

class Program

{

static void Main()

{

Library library = new Library();

// Adding books using integer indexer

library[101] = "C# Basics";

library[102] = "ASP.NET Core";

library[103] = "LINQ in Depth";

// Retrieving using Book ID

Console.WriteLine(library[101]);

Console.WriteLine(library[102]);

// Retrieving using Book Title

Console.WriteLine(library["ASP.NET Core"]);

// Book not found case

Console.WriteLine(library[999]);

}

}

## **3. Output (Expected)**

C# Basics

ASP.NET Core

ASP.NET Core

Book not found

## **4. Detailed Explanation**

### **Internal Data Storage**

private Dictionary<int, string> books;

* Stores **Book ID → Book Title**
* private ensures data hiding
* Indexers control all access

### **Integer-Based Indexer**

public string this[int bookId]

* Allows access using **numeric index**
* Supports both get and set
* Used to add or retrieve books

Usage:

library[101] = "C# Basics";

Console.WriteLine(library[101]);

This indexer behaves like a **dictionary wrapper**.

### **String-Based Indexer**

public string this[string title]

* Allows access using **book title**
* Read-only (no set)
* Searches values using LINQ
* Returns first matching title or null

Usage:

Console.WriteLine(library["ASP.NET Core"]);

## **5. What Is Indexer Overloading?**

**Indexer overloading** means:

Defining **multiple indexers in the same class** with **different parameter types**.

Here:

* this[int bookId]
* this[string title]

C# decides which indexer to use **based on parameter type**.

## **6. Why String-Based Indexer Is Read-Only**

* Writing by title can cause ambiguity
* Titles may not be unique
* IDs are safer for modification
* Prevents data inconsistency

## **7. Difference: Indexer Overloading vs Method Overloading**

| **Aspect** | **Indexer Overloading** | **Method Overloading** |
| --- | --- | --- |
| Syntax | obj[index] | obj.Method() |
| Purpose | Data access | Actions |
| Readability | High | Moderate |
| Use case | Collections | Operations |

## **8. Key Rules Demonstrated**

* Multiple indexers allowed
* Parameter types must differ
* Indexers improve readability
* Encapsulation is preserved
* LINQ enables flexible searching

## **9. Exam-Ready Definitions**

**Indexer Overloading:***Defining multiple indexers in a class with different parameter types to provide multiple ways of accessing the same object.*

## **10. One-Line Memory Rule**

**Different index types → different indexers → same object**

# 3. Inheritance in C#

## **3.1 Concept of Inheritance**

**Inheritance** allows one class to **reuse the properties and methods** of another class.

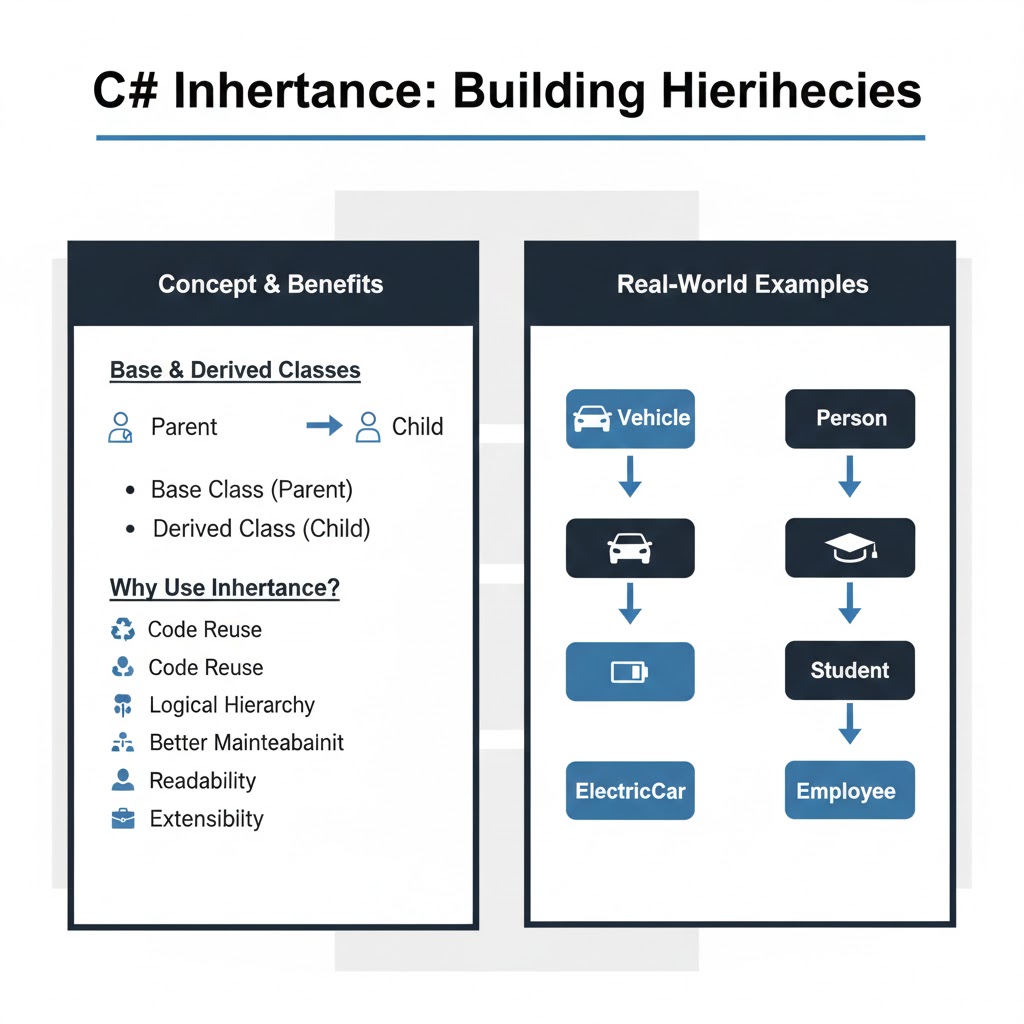
* The class being inherited from is called the **Base Class (Parent)**
* The class that inherits is called the **Derived Class (Child)**

### **Benefits of Inheritance**

* Code reuse (avoids duplication)
* Logical class hierarchy
* Better maintainability
* Improves readability
* Supports extensibility

### **Real-World Example**

* Vehicle → Car → ElectricCar
* Person → Student → Employee



## **3.2 Creating Inheritance**

Inheritance is created using the **colon (:)** symbol.

class Vehicle

{

public void Start()

{

Console.WriteLine("Vehicle started");

}

}

class Car : Vehicle

{

public void Drive()

{

Console.WriteLine("Car is driving");

}

}

### **Usage**

Car c = new Car();

c.Start(); // inherited method

c.Drive(); // own method

### **Output**

Vehicle started

Car is driving

## **Code Reusability:** By using the : symbol, the Car class automatically gains access to the Start() method from the Vehicle class, eliminating the need to rewrite common code.

## **Specialization:** The Car class "extends" the base class by adding its own unique behavior, the Drive() method, which is specific only to cars and not all vehicles.

## **Unified Access:** When you create a Car object, it acts as a combination of both classes, allowing you to call inherited methods (Start) and local methods (Drive) from a single instance.

## 

## **3.3 base Keyword**

The base keyword is used to access:

* Base class constructor
* Base class methods
* Base class variables

### **Calling Base Class Constructor**

class Person

{

public string Name;

public Person(string name)

{

Name = name;

}

}

class Student : Person

{

public int RollNo;

public Student(string name, int roll) : base(name)

{

RollNo = roll;

}

}

### **Usage**

Student s = new Student("Amit", 101);

Console.WriteLine(s.Name);

Console.WriteLine(s.RollNo);

**Output of the usage code:**

Amit

101

## **3.4 Types of Inheritance in C#**

### **Single Inheritance**

One base class, one derived class.

class Animal

{

public void Eat()

{

Console.WriteLine("Animal eats");

}

}

class Dog : Animal

{

public void Bark()

{

Console.WriteLine("Dog barks");

}

}

### **Multilevel Inheritance**

A class is derived from another derived class.

class LivingBeing

{

public void Breathe()

{

Console.WriteLine("Breathing");

}

}

class Human : LivingBeing

{

public void Think()

{

Console.WriteLine("Thinking");

}

}

class Employee : Human

{

public void Work()

{

Console.WriteLine("Working");

}

}

### **Hierarchical Inheritance**

Multiple classes inherit from one base class.

class Shape

{

public void Draw()

{

Console.WriteLine("Drawing shape");

}

}

class Circle : Shape { }

class Rectangle : Shape { }

### 

### **Multiple Inheritance**

C# **does not support multiple inheritance using classes** to avoid ambiguity.

// Not allowed

class A { }

class B { }

class C : A, B { } // Error

C# supports multiple inheritance using **interfaces**.

## **The Problem of Ambiguity (Diamond Problem):** If a class could inherit from two parents (Class A and Class B), and both parents had a method called Display(), the child class wouldn't know which version to execute. This confusion is known as the "Diamond Problem," and C# avoids it entirely by limiting a class to only one direct base class.

## **The Interface Solution:** While a class can only have one parent, it can "sign" as many contracts as it wants. Since Interfaces only define *what* a class should do (the signature) but not *how* to do it (the logic), there is no risk of conflicting code. This allows you to achieve the benefits of multiple inheritance safely and cleanly

## **The Interface Solution in C#**

### **Key Idea**

While C# restricts class inheritance, it **allows multiple interface inheritance**.

### **Why Interfaces Are Safe**

Interfaces:

* Define **what** a class should do
* Do **not** define **how** to do it (no implementation)

So there is:

* No conflicting logic
* No ambiguity

### **Example**

interface IPrintable

{

void Print();

}

interface IScannable

{

void Scan();

}

class Machine : IPrintable, IScannable

{

public void Print()

{

Console.WriteLine("Printing");

}

public void Scan()

{

Console.WriteLine("Scanning");

}

}

Here:

* Machine implements **multiple interfaces**
* No confusion because:
  + The class provides its own implementation

## **Real-Life Analogy (Very Important)**

### **Class Inheritance**

A person:

* Has **one biological parent** class

### **Interfaces**

A person can:

* Be a **Teacher**
* Be a **Driver**
* Be a **Speaker**

Multiple roles, no conflict.

## **Comparison Table (Exam-Friendly)**

| **Feature** | **Classes** | **Interfaces** |
| --- | --- | --- |
| Multiple inheritance | ❌ No | ✔ Yes |
| Method implementation | Yes | No (by default) |
| Ambiguity risk | High | None |
| Usage | Code reuse | Capability contract |

## **3.5 Method Overriding**

A derived class can **modify** the behavior of a base class method.

**Method overriding** allows a **derived (child) class** to **change or extend the behavior** of a method that is already defined in a **base (parent) class**.

In other words:

The child class provides its **own implementation** of a method inherited from the parent class.

## **Why Method Overriding Is Needed**

Method overriding is used when:

* The base class provides **general behavior**
* The derived class needs **specific behavior**

This supports **runtime polymorphism**.

### **Base Class**

class Animal

{

public virtual void Sound()

{

Console.WriteLine("Animal makes sound");

}

}

### **Derived Class**

class Dog : Animal

{

public override void Sound()

{

Console.WriteLine("Dog barks");

}

}

## **What is virtual in C#?**

### **Definition**

virtual is a keyword used in a **base (parent) class** to indicate that a method **can be overridden** by a derived (child) class.

### **Meaning in simple words**

virtual tells the compiler:  
 “This method is allowed to be changed by child classes.”

### **Example**

class Animal

{

public virtual void Sound()

{

Console.WriteLine("Animal makes sound");

}

}

### **Explanation**

* Sound() is a **virtual method**
* It provides **default behavior**
* Child classes may replace this behavior

Without virtual, overriding is **not allowed**.

## **What is override in C#?**

### **Definition**

override is a keyword used in a **derived (child) class** to **replace the behavior** of a virtual method defined in the base class.

### **Meaning in simple words**

override tells the compiler:  
 “I am providing my own version of the parent’s method.”

### **Example**

class Dog : Animal

{

public override void Sound()

{

Console.WriteLine("Dog barks");

}

}

### **Explanation**

* Sound() in Dog **replaces** Sound() in Animal
* Method signature must match exactly
* This enables **runtime polymorphism**

## **How virtual and override Work Together**

Animal a = new Dog();

a.Sound();

Output:

Dog barks

### **Why this happens**

* Reference type → Animal
* Object type → Dog
* At runtime, C# checks the **actual object**
* Calls the **overridden method**

This is called **runtime polymorphism**.

## **Why Both Keywords Are Required**

| **Keyword** | **Used In** | **Purpose** |
| --- | --- | --- |
| virtual | Base class | Allows overriding |
| override | Derived class | Replaces base behavior |

* virtual **permits** overriding
* override **performs** overriding

Both are mandatory.

## **3.6 base Keyword with Methods**

class Dog : Animal

{

public override void Sound()

{

base.Sound();

Console.WriteLine("Dog barks");

}

}

### **Output**

Animal makes sound

Dog barks

## **3.7 Access Modifiers and Inheritance**

| **Modifier** | **Accessible in Derived Class** |
| --- | --- |
| public | Yes |
| protected | Yes |
| private | No |
| internal | Yes (same assembly) |
| protected internal | Yes |

### 

### **Example**

class Base

{

protected int x = 10;

}

class Derived : Base

{

public void Show()

{

Console.WriteLine(x);

}

}

**Usage Code**

class Program

{

static void Main()

{

Derived d = new Derived();

d.Show(); // Output: 10

// Console.WriteLine(d.x); // ERROR: 'x' is protected and not accessible here

}

}

## **3.8 Constructor Execution Order**

When inheritance is used:

1. Base class constructor executes first
2. Derived class constructor executes next

class A

{

public A()

{

Console.WriteLine("A constructor");

}

}

class B : A

{

public B()

{

Console.WriteLine("B constructor");

}

}

### **Output**

A constructor

B constructor

## **3.9 Sealed Classes and Methods**

# **Sealed Classes and Sealed Methods in C#**

The **sealed keyword** in C# is used to **restrict inheritance and modification**.  
It tells the compiler:

“This is the final version. Do not extend or change it further.”

## **1. Sealed Class**

### **Definition**

A **sealed class** is a class that **cannot be inherited** by any other class.

Once a class is sealed:

* No derived (child) class can be created
* The logic inside the class cannot be extended or modified through inheritance

### **Code Example**

sealed class Security

{

}

### **Explanation**

* sealed prevents any class from inheriting Security
* The following code is **not allowed**:

class AdminSecurity : Security // Compile-time error

{

}

Error reason:

* C# does not allow inheritance from a sealed class

### **Why Sealed Classes Are Used**

1. **Security reasons**
   * Prevents malicious or accidental modification of critical logic
2. **Final business logic**
   * Ensures behavior remains unchanged
3. **Performance optimization**
   * CLR can optimize sealed classes because no inheritance is possible
4. **API design**
   * Prevents developers from misusing or extending sensitive classes

### **Real-World Examples**

* Authentication systems
* Encryption utilities
* Banking transaction validators
* Framework core classes

Example:

* Once a security algorithm is finalized, it should not be altered by inheritance

### **Important Rule**

A sealed class **cannot be inherited**, but it **can still be instantiated**.

## **2. Sealed Method**

### **Definition**

A **sealed method** is a method that **cannot be overridden further** in derived classes.

Important:

* A sealed method **must already be overridden**
* Sealing applies **only to overridden methods**

### **Code Example**

class Parent

{

public virtual void Show()

{

Console.WriteLine("Parent Show");

}

}

class Child : Parent

{

public sealed override void Show()

{

Console.WriteLine("Child Show");

}

}

### **Explanation (Step by Step)**

#### **class Parent**

* Base class
* Defines a virtual method Show()

#### **public virtual void Show()**

* virtual allows derived classes to override this method

#### **class Child : Parent**

* Child class inherits from Parent

#### **public sealed override void Show()**

* override → replaces parent implementation
* sealed → prevents further overriding

Meaning:

* Child can override Show()
* Any class inheriting from Child **cannot override Show() again**

### **What Is NOT Allowed**

class GrandChild : Child

{

public override void Show() // Compile-time error

{

}

}

Error reason:

* Show() is sealed in Child
* Further overriding is blocked

## **Why Sealed Methods Are Used**

1. **Lock specific behavior**
   * Prevents subclasses from changing critical logic
2. **Controlled extensibility**
   * Class can be extended, but some methods are fixed
3. **Security and consistency**
   * Ensures correct behavior in sensitive operations

## **Key Differences: Sealed Class vs Sealed Method**

| **Feature** | **Sealed Class** | **Sealed Method** |
| --- | --- | --- |
| Prevents inheritance | Yes | No |
| Prevents overriding | N/A | Yes |
| Can be instantiated | Yes | Yes |
| Used for | Final class | Final behavior |

## **Important Rules to Remember (Exam Gold)**

1. A sealed class **cannot be inherited**
2. A sealed method **must be overridden first**
3. Sealed methods work only with override
4. Sealed methods stop further overriding
5. sealed improves security and predictability

## **Exam-Ready Definitions**

### **Sealed Class**

**A sealed class is a class that cannot be inherited, ensuring its implementation remains final and unchanged.**

### **Sealed Method**

**A sealed method is an overridden method that cannot be further overridden in derived classes.**

## **One-Line Memory Rules**

* **sealed class → no child classes**
* **sealed method → no further overrides**

## **3.10 Inheritance vs Composition**

| **Inheritance** | **Composition** |
| --- | --- |
| IS-A relationship | HAS-A relationship |
| Tight coupling | Loose coupling |
| Reuse via hierarchy | Reuse via objects |

## **1. Inheritance**

### **Meaning**

**Inheritance** represents an **IS-A relationship**.

A child class **is a type of** parent class.

### **Key Idea**

* One class **inherits** properties and methods of another class
* Forms a **class hierarchy**

### **Example (Inheritance)**

class Vehicle

{

public void Move()

{

Console.WriteLine("Vehicle is moving");

}

}

class Car : Vehicle

{

}

Here:

* Car IS-A Vehicle
* Car automatically gets Move()

### **Characteristics of Inheritance**

1. **IS-A relationship**
   * Dog IS-A Animal
   * Car IS-A Vehicle
2. **Tight coupling**
   * Child class is strongly dependent on parent class
   * Changes in parent may affect child
3. **Reuse via hierarchy**
   * Code reuse happens through class inheritance tree
4. **Compile-time relationship**
   * Relationship is fixed at design time

### **Problems with Inheritance**

* Breaks easily when parent class changes
* Increases dependency
* Deep inheritance trees become hard to maintain
* Less flexible

## **2. Composition**

### **Meaning**

**Composition** represents a **HAS-A relationship**.

A class **contains** another class as a part of itself.

### **Key Idea**

* One class **uses** another class
* Objects are combined to build functionality

### **Example (Composition)**

class Engine

{

public void Start()

{

Console.WriteLine("Engine started");

}

}

class Car

{

Engine engine = new Engine();

public void Drive()

{

engine.Start();

Console.WriteLine("Car is driving");

}

}

Here:

* Car HAS-A Engine
* Car uses Engine but does not inherit from it

### **Characteristics of Composition**

1. **HAS-A relationship**
   * Car HAS-A Engine
   * House HAS-A Room
2. **Loose coupling**
   * Car depends on Engine behavior, not inheritance
   * Engine can be replaced without changing Car logic
3. **Reuse via objects**
   * Reuse achieved by creating and using objects
4. **Runtime flexibility**
   * Components can be swapped or modified at runtime

## **3. Side-by-Side Comparison (Very Important for Exams)**

| **Feature** | **Inheritance** | **Composition** |
| --- | --- | --- |
| Relationship | IS-A | HAS-A |
| Coupling | Tight | Loose |
| Flexibility | Low | High |
| Reuse method | Class hierarchy | Object composition |
| Change impact | High | Low |
| Recommended | Less | More (preferred) |

## **4. Why Composition Is Preferred Over Inheritance**

This is a **very important design principle**:

**“Favor composition over inheritance.”**

### **Reasons:**

* Reduces dependency
* Easier to maintain
* More flexible design
* Avoids fragile base class problem

## **5. Real-World Analogy (Very Clear)**

### **Inheritance**

* Student IS-A Person
* Dog IS-A Animal

### **Composition**

* Car HAS-A Engine
* Laptop HAS-A Battery
* Company HAS-A Employees

You **cannot inherit an engine**, you **use** it.

## **6. When to Use What**

### **Use Inheritance when:**

* Relationship is truly IS-A
* Behavior should be shared permanently
* Polymorphism is required

### **Use Composition when:**

* Relationship is HAS-A
* Behavior may change
* You want flexibility and loose coupling

## **7. Exam-Ready Paragraph Answer**

**Inheritance represents an IS-A relationship where a class derives behavior from another class, leading to tight coupling and reuse through class hierarchy. Composition represents a HAS-A relationship where a class contains objects of other classes, resulting in loose coupling and reuse through object composition. Modern object-oriented design prefers composition over inheritance due to better flexibility and maintainability.**

## **One-Line Memory Rule**

**Inheritance = IS-A (tight)  
Composition = HAS-A (loose)**

## **3.11 Real-World Use Cases**

* Employee management systems
* Banking applications
* Game development (Player, Enemy, NPC)
* UI frameworks

## **4. Method Hiding vs Method Overriding**

Both **method hiding** and **method overriding** allow a derived class to define a method with the same name as the base class method, but **their behavior and purpose are very different**.

## **4.1 Method Hiding (new Keyword)**

Method hiding occurs when a **derived class defines a method with the same name** as a method in the base class **without virtual/override**, using the new keyword.

class Parent

{

public void Show()

{

Console.WriteLine("Parent Show");

}

}

class Child : Parent

{

public new void Show()

{

Console.WriteLine("Child Show");

}

}

### **Key Characteristics of Method Hiding**

* Decided at compile time
* Depends on reference type, not object type
* Does not support runtime polymorphism
* Base class method is not overridden, only hidden

### **Example: Reference Type Dependency**

Parent p = new Child();

p.Show();

Child c = new Child();

c.Show();

### **Output**

Parent Show

Child Show

Explanation:

* p.Show() calls Parent method because reference type is Parent
* c.Show() calls Child method because reference type is Child

### **When to Use Method Hiding**

* When base class method cannot be changed
* When you want a different implementation without affecting base behavior
* Rarely used in real-world applications

## **4.2 Method Overriding (virtual & override)**

Method overriding allows a **derived class to replace the behavior** of a base class method.

### **Base Class**

class Animal

{

public virtual void Speak()

{

Console.WriteLine("Animal sound");

}

}

### **Derived Class**

class Dog : Animal

{

public override void Speak()

{

Console.WriteLine("Dog barks");

}

}

### **Key Characteristics of Method Overriding**

* Decided at **runtime**
* Depends on **object type**
* Supports **runtime polymorphism**
* Base method must be virtual
* Derived method must use override

### **Example: Runtime Polymorphism**

Animal a = new Dog();

a.Speak();

### 

### **Output**

Dog barks

Explanation:

* Even though reference type is Animal, object is Dog
* Runtime decides which method to call

## **4.3 Method Hiding vs Overriding (Comparison Table)**

| **Feature** | **Method Hiding** | **Method Overriding** |
| --- | --- | --- |
| Keyword | new | virtual & override |
| Binding | Compile time | Runtime |
| Polymorphism | Not supported | Supported |
| Reference Based | Yes | No |
| Object Based | No | Yes |
| Base Method | Not replaced | Replaced |
| Usage | Rare | Very common |

## **4.4 base Keyword with Overriding**

The base keyword can be used to call the base class method from an overridden method.

class Dog : Animal

{

public override void Speak()

{

base.Speak();

Console.WriteLine("Dog barks");

}

}

### **Output**

Animal sound

Dog barks

## **4.5 Can Method Hiding Be Done Without new?**

Yes, but the compiler gives a warning.

class Child : Parent

{

public void Show()

{

Console.WriteLine("Child Show");

}

}

Compiler warning:

'Child.Show()' hides inherited member 'Parent.Show()'

Best practice: always use new explicitly.

## 

## **4.6 Can Static Methods Be Overridden?**

No. Static methods **cannot be overridden**, but they **can be hidden**.

class A

{

public static void Display()

{

Console.WriteLine("A Display");

}

}

class B : A

{

public new static void Display()

{

Console.WriteLine("B Display");

}

}

## **4.7 Sealed Override**

A derived class can prevent further overriding using sealed.

class Dog : Animal

{

public sealed override void Speak()

{

Console.WriteLine("Dog barks");

}

}

Further inheritance attempting override will cause an error.

## **4.8 Real-World Use Cases**

### **Method Hiding**

* Legacy systems
* Backward compatibility
* Framework extensions (rare)

### **Method Overriding**

* UI frameworks
* Game engines
* Web APIs
* Business rule customization

## **4.9 One-Line Summary**

* Method hiding depends on **reference type**
* Method overriding depends on **object type**
* Use **override** for polymorphism
* Use **new** only when necessary

## **5. Sealed Classes & Methods**

The sealed keyword in C# is used to **restrict inheritance and overriding**, helping to protect critical logic and improve performance and security.

## **5.1 Sealed Class**

A **sealed class** is a class that **cannot be inherited**.

sealed class SecurityManager

{

public void Authenticate()

{

Console.WriteLine("Authentication successful");

}

}

Attempting to inherit from a sealed class results in a **compile-time error**.

class AdvancedSecurity : SecurityManager // Error

{

}

### **Key Characteristics of Sealed Classes**

* Prevents further inheritance
* Final level in class hierarchy
* Improves security
* Prevents misuse or alteration of logic
* Slight performance benefit (no virtual dispatch)

### **When to Use Sealed Classes**

* Security-related components
* Utility/helper classes
* Logging frameworks
* Core business rules
* Payment gateways
* Authentication systems

### **Real-World Example**

sealed class PaymentProcessor

{

public void ProcessPayment()

{

Console.WriteLine("Payment processed");

}

}

## **5.2 Sealed Method**

A **sealed method** prevents further overriding of a method **after it has already been overridden**.

Important rule: Only an **overridden method** can be sealed.

### **Example**

class Bank

{

public virtual void CalculateInterest()

{

Console.WriteLine("Base interest");

}

}

class SBI : Bank

{

public sealed override void CalculateInterest()

{

Console.WriteLine("SBI interest logic");

}

}

Now, any further derived class **cannot override** CalculateInterest.

class Branch : SBI

{

public override void CalculateInterest() // Error

{

}

}

### **Key Characteristics of Sealed Methods**

* Used with override
* Prevents further overriding
* Maintains consistency in derived classes
* Allows inheritance but locks specific behavior

## **5.3 Why Sealed Methods Are Needed**

* Prevent accidental logic changes
* Protect financial or regulatory rules
* Ensure consistent behavior across subclasses
* Improve maintainability
* Avoid unpredictable runtime behavior

## **5.4 Sealed Class vs Sealed Method**

| **Feature** | **Sealed Class** | **Sealed Method** |
| --- | --- | --- |
| Prevents Inheritance | Yes | No |
| Prevents Overriding | All methods | Only specific method |
| Used With | class | override |
| Flexibility | Low | Medium |
| Common Usage | Utility, security | Business rules |

## **5.5 Can a Sealed Class Have Virtual Methods?**

No.

sealed class Test

{

public virtual void Show() // Compile-time error

{

}

}

Reason:

* Virtual methods exist to be overridden
* Sealed classes cannot be inherited

## **5.6 Can Sealed Methods Be Hidden?**

Yes. A sealed method **cannot be overridden**, but it **can be hidden** using the new keyword.

class Branch : SBI

{

public new void CalculateInterest()

{

Console.WriteLine("Branch interest");

}

}

This is method hiding, not overriding.

## **5.7 Sealed Classes and Performance**

* Sealed classes allow the CLR to make optimizations
* No virtual method lookup needed
* Faster method invocation

This is why many .NET framework classes are sealed.

## **5.8 Sealed Keyword and Interfaces**

* Interfaces **cannot** be sealed
* Classes implementing interfaces **can** be sealed

interface ILogger

{

void Log();

}

sealed class FileLogger : ILogger

{

public void Log()

{

Console.WriteLine("Logging to file");

}

}

## **5.9 Sealed vs Abstract**

| **Feature** | **Sealed** | **Abstract** |
| --- | --- | --- |
| Can be inherited | No | Yes |
| Can have abstract methods | No | Yes |
| Object creation | Allowed | Not allowed |
| Purpose | Restrict | Enforce |

# **1-Hour Assignment**

## Insurance Management System (OOP Based)

## Below is a clean, professional, exam / lab–ready PROBLEM STATEMENT for the given C# program, followed by the Expected Output. This is written exactly in the style teachers expect for practical exams, assignments, or viva questions.

## 

# Problem Statement

## Title: Insurance Management System using Advanced OOP Concepts in C#

## 

## Objective

## Design and implement a console-based Insurance Management System in C# that demonstrates the use of advanced object-oriented programming concepts including:

## Sealed classes

## Abstract base classes

## Inheritance and method overriding

## Method hiding using new keyword

## Init-only properties

## Encapsulation with validation

## Indexers

## Runtime polymorphism

## 

## Problem Description

## You are required to build an Insurance System for an insurance company that manages different types of insurance policies. The system must follow real-world business rules and strict object-oriented design principles.

## 

## System Requirements

### 1. Security Module

## Create a sealed class that handles user authentication.

## This class must not be inherited.

## It should display a confirmation message after authentication.

## 

### 2. Base Insurance Policy

## Create an abstract base class named InsurancePolicy.

## The class should contain:

## An init-only property for Policy Number

## A validated premium property (premium must be greater than zero)

## A policy holder name

## Include:

## A virtual method to calculate the premium

## A method to display generic policy information

## 

### 3. Derived Insurance Types

#### a) Life Insurance

## Inherit from InsurancePolicy

## Override the premium calculation method by adding a fixed life-insurance charge

## Demonstrate method hiding by redefining the policy display method

#### b) Health Insurance

## Inherit from InsurancePolicy

## Override the premium calculation method

## Seal the overridden method to prevent further modification

## 

### 4. Policy Directory (Indexer Implementation)

## Create a class that stores multiple insurance policies.

## Use a List internally to store policies.

## Implement indexers to:

## Access policies by index

## Access policies by policy holder name

## 

### 5. Main Program Execution

## The main program should:

## Authenticate the user

## Create life and health insurance policies

## Store them in the policy directory

## Retrieve policy details using indexers

## Demonstrate:

## Runtime polymorphism using overridden methods

## Method hiding using base and derived class references

## 

## Expected Output

## User authenticated successfully

## 

## Amit

## 102

## Life Premium: 5500

## Health Premium: 8000

## Life Insurance Policy

## Insurance Policy

## 

## 

## Explanation of Output

## Authentication message

## Displayed by the sealed security class.

## Indexer by index

## Displays the holder name of the first policy (Amit).

## Indexer by name

## Displays the policy number of the policy holder named Neha.

## Runtime polymorphism

## Correct premium calculation based on actual object type.

## Method hiding demonstration

## Calling ShowPolicy() using a derived reference prints: Life Insurance Policy

## 

## Calling the same method using a base-class reference prints: Insurance Policy

## 

## 

## 

## 

## **Complete C# Code**

using System;

using System.Collections.Generic;

using System.Linq;

namespace InsuranceSystem

{

// -----------------------------

// SEALED CLASS (Security Module)

// -----------------------------

sealed class SecurityModule

{

public void Authenticate()

{

Console.WriteLine("User authenticated successfully\n");

}

}

// -----------------------------

// BASE CLASS (Inheritance)

// -----------------------------

abstract class InsurancePolicy

{

// Auto-implemented property

public int PolicyNo { get; init; }

// Read/Write property with validation

private double basePremium;

public double BasePremium

{

get { return basePremium; }

protected set

{

if (value > 0)

basePremium = value;

}

}

public string HolderName { get; set; }

protected InsurancePolicy(int policyNo, string holder, double premium)

{

PolicyNo = policyNo;

HolderName = holder;

BasePremium = premium;

}

// Virtual method

public virtual double CalculatePremium()

{

return BasePremium;

}

// Method to demonstrate hiding

public void ShowPolicy()

{

Console.WriteLine("Insurance Policy");

}

}

// --------------------------------

// DERIVED CLASS – LIFE INSURANCE

// --------------------------------

class LifeInsurance : InsurancePolicy

{

public LifeInsurance(int no, string holder, double premium)

: base(no, holder, premium)

{

}

public override double CalculatePremium()

{

return base.CalculatePremium() + 500;

}

// Method Hiding

public new void ShowPolicy()

{

Console.WriteLine("Life Insurance Policy");

}

}

// --------------------------------

// DERIVED CLASS – HEALTH INSURANCE

// --------------------------------

class HealthInsurance : InsurancePolicy

{

public HealthInsurance(int no, string holder, double premium)

: base(no, holder, premium)

{

}

// Sealed override method

public sealed override double CalculatePremium()

{

return base.CalculatePremium() + 1000;

}

}

// --------------------------------

// INDEXER CLASS (Policy Collection)

// --------------------------------

class PolicyDirectory

{

private List<InsurancePolicy> policies = new List<InsurancePolicy>();

// Indexer by index

public InsurancePolicy this[int index]

{

get { return policies[index]; }

}

// Indexer by holder name

public InsurancePolicy this[string name]

{

get

{

return policies.FirstOrDefault(p => p.HolderName == name);

}

}

public void AddPolicy(InsurancePolicy policy)

{

policies.Add(policy);

}

}

// -----------------------------

// MAIN PROGRAM

// -----------------------------

class Program

{

static void Main()

{

// Sealed class usage

SecurityModule security = new SecurityModule();

security.Authenticate();

// Create policies

InsurancePolicy p1 = new LifeInsurance(101, "Amit", 5000);

InsurancePolicy p2 = new HealthInsurance(102, "Neha", 7000);

// Indexer usage

PolicyDirectory directory = new PolicyDirectory();

directory.AddPolicy(p1);

directory.AddPolicy(p2);

// Access using indexer

Console.WriteLine(directory[0].HolderName);

Console.WriteLine(directory["Neha"].PolicyNo);

// Method overriding (runtime polymorphism)

Console.WriteLine("Life Premium: " + p1.CalculatePremium());

Console.WriteLine("Health Premium: " + p2.CalculatePremium());

// Method hiding

LifeInsurance life = new LifeInsurance(103, "Rahul", 6000);

life.ShowPolicy();

InsurancePolicy baseRef = life;

baseRef.ShowPolicy();

}

}

}

## 

## **Expected Output**

User authenticated successfully

Amit

102

Life Premium: 5500

Health Premium: 8000

Life Insurance Policy

Insurance Policy

Insurance

## Insurance Management System (OOP Based)

## 

# Problem Statement

## Title: Insurance Management System using Advanced OOP Concepts in C#

## 

## Objective

## Design and implement a console-based Insurance Management System in C# that demonstrates the use of advanced object-oriented programming concepts including:

## Sealed classes

## Abstract base classes

## Inheritance and method overriding

## Method hiding using new keyword

## Init-only properties

## Encapsulation with validation

## Indexers

## Runtime polymorphism

## 

## Problem Description

## You are required to build an Insurance System for an insurance company that manages different types of insurance policies. The system must follow real-world business rules and strict object-oriented design principles.

## 

## System Requirements

### 1. Security Module

## Create a sealed class that handles user authentication.

## This class must not be inherited.

## It should display a confirmation message after authentication.

## 

### 2. Base Insurance Policy

## Create an abstract base class named InsurancePolicy.

## The class should contain:

## An init-only property for Policy Number

## A validated premium property (premium must be greater than zero)

## A policy holder name

## Include:

## A virtual method to calculate the premium

## A method to display generic policy information

## 

### 3. Derived Insurance Types

#### a) Life Insurance

## Inherit from InsurancePolicy

## Override the premium calculation method by adding a fixed life-insurance charge

## Demonstrate method hiding by redefining the policy display method

#### b) Health Insurance

## Inherit from InsurancePolicy

## Override the premium calculation method

## Seal the overridden method to prevent further modification

## 

### 4. Policy Directory (Indexer Implementation)

## Create a class that stores multiple insurance policies.

## Use a List internally to store policies.

## Implement indexers to:

## Access policies by index

## Access policies by policy holder name

## 

### 5. Main Program Execution

## The main program should:

## Authenticate the user

## Create life and health insurance policies

## Store them in the policy directory

## Retrieve policy details using indexers

## Demonstrate:

## Runtime polymorphism using overridden methods

## Method hiding using base and derived class references

## 

## Expected Output

## User authenticated successfully

## 

## Amit

## 102

## Life Premium: 5500

## Health Premium: 8000

## Life Insurance Policy

## Insurance Policy

## 

## 

## Explanation of Output

## Authentication message

## Displayed by the sealed security class.

## Indexer by index

## Displays the holder name of the first policy (Amit).

## Indexer by name

## Displays the policy number of the policy holder named Neha.

## Runtime polymorphism

## Correct premium calculation based on actual object type.

## Method hiding demonstration

## Calling ShowPolicy() using a derived reference prints: Life Insurance Policy

## 

## Calling the same method using a base-class reference prints: Insurance Policy

## 

Extended Task - Insurance

# **EXTENDED TASK – INSURANCE SYSTEM (C#)**

## **Objective**

Extend the existing **Insurance Management System** by implementing **missing functionalities** related to **indexers, inheritance, and sealed behavior**, while preserving the original design.

## **Precondition**

You are given an existing Insurance System that already includes:

* Abstract base class InsurancePolicy
* Derived classes LifeInsurance and HealthInsurance
* Sealed class SecurityModule
* Indexer-based PolicyDirectory
* Init-only properties and overridden methods

You must **extend the system without breaking existing code**.

## **TASK 1 – Indexer Enhancement (NOT IMPLEMENTED)**

### **Task 1.1: Add Indexer to Search by Policy Number**

**Requirement**Allow policies to be searched using their **Policy Number**.

**Task Instruction**

* Add a new indexer in PolicyDirectory
* The indexer should:
  + Accept an integer policy number
  + Return the matching InsurancePolicy object

**Example (Conceptual)**

directory[101]

### **Task 1.2: Handle Invalid Index Safely**

**Requirement**Prevent runtime exceptions when:

* An invalid index is used
* A policy number does not exist
* A policy holder name is not found

**Task Instruction**

* Modify existing indexers so that:
  + No IndexOutOfRangeException occurs
  + No unhandled exceptions are thrown
  + A safe value (null or meaningful message) is returned

**Concept Tested**

* Defensive programming
* Robust indexer design

## **TASK 2 – Inheritance Extension (NOT IMPLEMENTED)**

### **Task 2.1: Add VehicleInsurance Class**

**Requirement**Introduce a new insurance type called **Vehicle Insurance**.

**Task Instruction**

* Create a class VehicleInsurance
* Inherit it from InsurancePolicy
* Follow the same constructor pattern as other insurance types

**Concept Tested**

* Inheritance
* Code reuse

### **Task 2.2: Override Premium Calculation**

**Requirement**Vehicle insurance must calculate premium differently.

**Task Instruction**

* Override the premium calculation method
* Add vehicle-specific charges
* Ensure runtime polymorphism works correctly

**Concept Tested**

* Method overriding
* Polymorphism

## **TASK 3 – Sealed Method Restriction (PARTIALLY IMPLEMENTED)**

### **Task 3.1: Attempt to Override a Sealed Method**

**Requirement**Demonstrate the effect of sealing a method.

**Task Instruction**

* Create a class that attempts to inherit from HealthInsurance
* Try overriding the sealed premium calculation method
* Observe and document the compiler error

**Expected Observation**

* Code should fail at compile time
* Error message should indicate that overriding a sealed method is not allowed

**Concept Tested**

* Sealed methods
* Compile-time enforcement

## **TASK 4 – Integration Test (NEW)**

### **Task 4.1: Integrate New Insurance Type**

**Requirement**Include the newly created VehicleInsurance in the main program.

**Task Instruction**

* Create at least one VehicleInsurance object
* Add it to PolicyDirectory
* Access it using the new policy-number-based indexer
* Display its calculated premium

## **Constraints**

* Do not modify existing implemented features
* Do not remove sealed behavior
* Follow existing naming conventions
* Use proper access modifiers
* No direct access to internal collections

## **Learning Outcomes**

After completing this extended task, students will be able to:

* Design **safe and flexible indexers**
* Extend systems using **inheritance without breaking base code**
* Understand **why sealed methods cannot be overridden**
* Apply **runtime polymorphism** across multiple derived classes
* Build **robust, extensible object-oriented systems**

## **Evaluation Hint (For Exams / Labs)**

Marks should be awarded for:

* Correct indexer logic
* Safe error handling
* Proper inheritance structure
* Correct use of sealed restrictions
* Clean and maintainable code

### **Memory Rule**

**Extend behavior → override  
Restrict behavior → seal  
Access data → indexer**