Object Oriented Programming Part2

1. Types of Attributes (Fields)

Attributes are variables declared inside a class. They represent the state or data of an object. You can classify them by:

🔷 a. By Access Modifier

- **Public:** Accessible from anywhere.
- **Private:** Accessible only within the class.
- **Protected:** Accessible within the class and subclasses.
- **Internal:** Accessible within the same assembly.
- **Protected Internal:** Accessible within the same assembly or any derived class.

```
class Person
{
  public string Name;        // Accessible everywhere
  private int age;           // Only accessible within this class
  protected string Gender;   // Accessible in derived classes
  internal string NationalID; // Accessible within the same assembly
}
```

🔷 b. By Behavior

- **Instance Attributes:** Belong to an instance of a class.
- **Static Attributes:** Shared across all instances.

```
class Student
{
  public string Name;        // Instance attribute
  public static int Count;   // Static attribute shared across all students

  public Student()
  {
    Count++; // Tracks number of instances created
  }
}
```

2. Types of Methods

🔷 a. Instance Methods

Operate on a specific instance of a class.

```
class Calculator
{
    public int Add(int a, int b)
    {
        return a + b;
    }
}
```

🔷 b. Static Methods

Belong to the class, not the object. Can't access instance variables directly.

```
class MathUtils
{
    public static double Square(double x)
    {
        return x * x;
    }
}
```

🔷 c. Accessor Methods (Getters and Setters)

Used to **encapsulate** private fields.

```
class Employee
{
    private double salary;

    public double GetSalary()
    {   return salary;   }

    public void SetSalary(double value)
    {    if (value >= 0)   salary = value; }
}
```

Or more commonly using **properties**

In C#, **properties** provide a **flexible mechanism** to read, write, or compute the value of a private field. They are used instead of traditional getter and setter methods.

A property wraps around a **field** and contains two accessors:

- get: returns the value.
- set: assigns a value.

```
class Person
{
   private string name; // private field

   public string Name   // property
   {
      get { return name; }
      set { name = value; }
   }
}
//How to use
Person p = new Person();
p.Name = "Karim";      // Calls set accessor
Console.WriteLine(p.Name);  // Calls get accessor
```

Why Use Properties?

- Control how values are accessed or modified.
- Apply validation logic in set.
- Allow read-only or write-only access.
- Hide internal data structures.

Types of Properties

1. ✅ Auto-Implemented Properties

When no extra logic is needed in get or set, use this:

```
public string Email { get; set; }
```

The compiler creates a hidden **backing field** automatically.

2. ✅ Read-Only Property

You can expose only the get accessor.

```
private int age = 25;

public int Age

{

   get { return age; }

}
```

---

3. ✅ Write-Only Property

Useful when data should be written but not read.

```
private string password;

public string Password

{

   set { password = value; }

}
```

---

4. ✅ Property with Validation Logic

```
private int salary;

public int Salary

{

   get { return salary; }

   set

   {

      if (value >= 0)

         salary = value;

      else

         Console.WriteLine("Salary can't be negative");

   }

}
```

Full Example

```
class Product
{
    private double price;
    public string Name { get; set; } // Auto-property
    public double Price
    {
        get { return price; }
        set
        {
            if (value > 0)
                price = value;
            else
                Console.WriteLine("Price must be positive");
        }
    }
    public string Description { get; } = "Default product";
}
```

Usage:

```
csharp
CopyEdit
Product p = new Product();
p.Name = "Laptop";
p.Price = -1000; // Will show error
p.Price = 2000;

Console.WriteLine(p.Name);      // Laptop
Console.WriteLine(p.Price);     // 2000
Console.WriteLine(p.Description); // Default product
```

🔷 d. Constructors

Special method used to initialize objects.

```
class Book
{
   public string Title;
   public Book(string title)
   {     Title = title;    }
}
```

🔷 e. Destructors

Called when an object is destroyed (rarely needed in C# due to garbage collection).

```
class Resource
{
   ~Resource()
   {  // Clean up  }
}
```

Encapsulation

Definition: Wrapping data (attributes) and methods into a single unit (class) and restricting access to internal details using access modifiers.

🔷 Goal:

- Hide the internal implementation.
- Expose only what's necessary.

🔷 Example:

```
class BankAccount
{
   private double balance;
   public void Deposit(double amount)
   {  if (amount > 0)   balance += amount;  }
   public double GetBalance()
   {    return balance;    }
}
```

✅ **Summary Table**

Concept	**Description**	**Code Example**
Public Field	Visible everywhere	public string Name;
Private Field	Hidden from outside	private int age;
Static Field	Shared across objects	public static int Count;
Instance Method	Works on object	public int Add(...)
Static Method	Works on class	public static void Print()
Constructor	Initializes object	public Person(string name)
Getter/Setter	Access private fields	public string Name { get; set; }
Encapsulation	Hide fields, expose methods	private double balance; public void Deposit()