Part1

```
using System;

public interface IVehicle
{
   void StartEngine();
   void StopEngine();
}

public class Car : IVehicle
{
   public void StartEngine()
   {
      Console.WriteLine("Car engine started.");
   }

   public void StopEngine()
   {
      Console.WriteLine("Car engine stopped.");
   }
}

public class Bike : IVehicle
{
   public void StartEngine()
   {
      Console.WriteLine("Bike engine started.");
   }
```

```
    public void StopEngine()
    {
        Console.WriteLine("Bike engine stopped.");
    }
}
public class Program
{
    public static void Main()
    {
        IVehicle myCar = new Car();
        IVehicle myBike = new Bike();

        myCar.StartEngine();
        myCar.StopEngine();

        myBike.StartEngine();
        myBike.StopEngine();
    }
}
```

- أن الـ **Interface** **بيحقق المرونة** ← لو بكرة عاوز تضيف مركبة جديدة (زي `Bus` مش هتغير الكود اللي بيتعامل مع `IVehicle`.
- بيخلي الكود **قابل للتوسّع (Extensible)** ومش مربوط بفئة معينة.
- بيساعد على **الـ Polymorphism** (تعدد الأشكال) ← نفس الكود يشتغل مع أنواع مختلفة.
- بيخلي المشروع **سهل الصيانة والاختبار**.

```
using System;
```

```
public abstract class Shape
{
    public abstract double GetArea();
    public void Display()
    {
        Console.WriteLine("This is a shape.");
    }
}

public class Rectangle : Shape
{
    private double width, height;
    public Rectangle(double w, double h)
    {
        width = w; height = h;
    }
    public override double GetArea()
    {
        return width * height;
    }
}

public class Circle : Shape
{
    private double radius;
    public Circle(double r)
    {
        radius = r;
    }
    public override double GetArea()
    {
        return Math.PI * radius * radius;
    }
}

public interface IShape
{
    double GetArea();
    void Display();
}

public class RectangleInterface : IShape
{
    private double width, height;
    public RectangleInterface(double w, double h)
    {
        width = w; height = h;
    }
    public double GetArea()
    {
        return width * height;
    }
    public void Display()
    {
        Console.WriteLine("This is a rectangle.");
    }
}

public class CircleInterface : IShape
{
    private double radius;
    public CircleInterface(double r)
    {
        radius = r;
    }
    public double GetArea()
    {
        return Math.PI * radius * radius;
    }
```

```
    public void Display()
    {
        Console.WriteLine("This is a circle.");
    }
}

public class Program
{
    public static void Main()
    {
        Shape rect = new Rectangle(5, 10);
        Shape circle = new Circle(7);
        Console.WriteLine(rect.GetArea());
        Console.WriteLine(circle.GetArea());

        IShape rectInt = new RectangleInterface(5, 10);
        IShape circleInt = new CircleInterface(7);
        Console.WriteLine(rectInt.GetArea());
        Console.WriteLine(circleInt.GetArea());
    }
}
```

- ستخدم **interface**لو عايز **تحديد سلوك/عقد (Contract)** من غير تفاصيل.
- استخدم **abstract class**لو عايز **أساس مشترك فيه كود جاهز + مرونة للتوسع**.

```
using System;


public class Product : IComparable<Product>
{
    public int Id { get; set; }
    public string Name { get; set; }
    public double Price { get; set; }


    public Product(int id, string name, double price)
    {
        Id = id;
        Name = name;
        Price = price;
```

```
    }

    public int CompareTo(Product other)
    {
        return this.Price.CompareTo(other.Price);
    }

    public override string ToString()
    {
        return $"Id: {Id}, Name: {Name}, Price: {Price}";
    }
}

public class Program
{
    public static void Main()
    {
        Product[] products =
        {
            new Product(1, "Laptop", 1200),
            new Product(2, "Phone", 800),
            new Product(3, "Tablet", 600)
        };
```

```
        Array.Sort(products);


        foreach (var p in products)
        {
            Console.WriteLine(p);
        }
    }
}
```

- بدل ما نكتب كود Sorting مخصص كل مرة، نقدر نخلي الكلاس نفسه يعرف **إزاي يتقارن**.
- أي مجموعة (Array, List) من المنتجات تقدر تتفرز بسهولة بـ `Sort()` من غير ما نكتب Logic زيادة.
- لو غيرنا معيار المقارنة) مثلاً بالـ Name بدل (Price نعدل في `CompareTo` مرة واحدة بس.
- بيخلي الكود **قابل لإعادة الاستخدام** و **مرن للتغييرات**.

```
using System;

public class Student
{
    public int Id { get; set; }
    public string Name { get; set; }
    public Grade Grade { get; set; }
```

```
    public Student(int id, string name, Grade grade)
    {
        Id = id;
        Name = name;
        Grade = grade;
    }

    public Student(Student other)
    {
        Id = other.Id;
        Name = other.Name;
        Grade = new Grade(other.Grade.Score); // Deep Copy للـ object الداخلي
    }

    public override string ToString()
    {
        return $"Id: {Id}, Name: {Name}, Grade: {Grade.Score}";
    }
}

public class Grade
{
    public int Score { get; set; }
```

```
    public Grade(int score)
    {
        Score = score;
    }
}

public class Program
{
    public static void Main()
    {
        Student s1 = new Student(1, "Ali", new Grade(90));

        Student shallowCopy = s1;

        Student deepCopy = new Student(s1);

        Console.WriteLine("Before change:");
        Console.WriteLine($"Original: {s1}");
        Console.WriteLine($"Shallow: {shallowCopy}");
        Console.WriteLine($"Deep: {deepCopy}");

        s1.Grade.Score = 50;
```

```
        Console.WriteLine("\nAfter change:");
        Console.WriteLine($"Original: {s1}");
        Console.WriteLine($"Shallow: {shallowCopy}");
        Console.WriteLine($"Deep: {deepCopy}");
    }
}
```

- الهدف الأساسي إنه يعمل **نسخة مستقلة (Deep Copy)** من الكائن.
- يعني الكائن الجديد مايشاركش نفس المراجع (references) مع الكائن الأصلي.
- ده بيوفر **عزل تام** بحيث أي تعديل على نسخة مايأثرش على التانية.

```
using System;

public interface IWalkable
{
    void Walk();
}

public class Robot : IWalkable
{
    public void Walk()
    {
        Console.WriteLine("Robot walking normally.");
    }
```

```
    void IWalkable.Walk()
    {
        Console.WriteLine("Robot walking as IWalkable.");
    }
}

public class Program
{
    public static void Main()
    {
        Robot r = new Robot();
        r.Walk();

        IWalkable iw = r;
        iw.Walk();
    }
```

- عندك method في الكلاس لها نفس الاسم زي اللي في الـinterface ، ممكن يحصل تضارب.
- باستخدام **explicit implementation** بتقدر تميز بين استدعاء الـ method من الكلاس نفسه وبين استدعاءها من خلال الـinterface.
- ده بيدي **تحكم كامل** في سلوك كل حالة ويمنع أي تعارض في الأسماء.

```
using System;

public struct Account
```

```
{
    private int accountId;
    private string accountHolder;
    private double balance;

    public int AccountId
    {
        get { return accountId; }
        set { accountId = value; }
    }

    public string AccountHolder
    {
        get { return accountHolder; }
        set { accountHolder = value; }
    }

    public double Balance
    {
        get { return balance; }
        set { balance = value; }
    }
```

```
    public Account(int id, string holder, double bal)
    {
        accountId = id;
        accountHolder = holder;
        balance = bal;
    }

    public override string ToString()
    {
        return $"Id: {AccountId}, Holder: {AccountHolder}, Balance: {Balance}";
    }
}

public class Program
{
    public static void Main()
    {
        Account acc = new Account(101, "Ali", 5000);
        Console.WriteLine(acc);

        acc.Balance = 7000;
        Console.WriteLine(acc);
    }
```

}

- **struct**نوع → **Value Type**بيتخزن في **الـ Stack** (عادةً)، والنسخ منه بتكون نسخ مستقلة.
- **class**نوع → **Reference Type**بيتخزن في **Heap**، والنسخ منه بتشير لنفس الكائن.
- لكن الاتنين بيدعموا) **Encapsulation**إخفاء البيانات + خصائص للوصول.(
- الفكرة إن الفرق الأساسي هو **طريقة التخزين والإدارة في الذاكرة**، مش في مبدأ الـ encapsulation نفسه.

```
using System;


public interface ILogger
{
   void Log(string message)
   {
      Console.WriteLine("Default log: " + message);
   }
}


public class ConsoleLogger : ILogger
{
   public void Log(string message)
   {
      Console.WriteLine("Console log: " + message);
   }
}
```

```
public class Program
{
    public static void Main()
    {
        ILogger logger1 = new ConsoleLogger();
        logger1.Log("Hello from ConsoleLogger");

        ILogger logger2 = new TestLogger();
        logger2.Log("Hello from TestLogger");
    }
}

public class TestLogger : ILogger
{
}
```

- قبل C# 8: أي إضافة method جديدة لـ interface كانت **تكسر الكود القديم**، لأن كل الكلاسات اللي بتطبق الـ interface لازم تنفذها.
- مع **default implementations**: تقدر تضيف method جديدة وتديها **سلوك افتراضي** → الكلاسات القديمة مش محتاجة تتعدل.
- ده بيخلي الكود أكثر **مرونة** ويمنع **كسر التطبيقات القديمة** عند تطوير الـ interface.

```
using System;

public class Book
```

```
{
    public string Title { get; set; }
    public string Author { get; set; }

    public Book()
    {
        Title = "Unknown";
        Author = "Unknown";
    }

    public Book(string title)
    {
        Title = title;
        Author = "Unknown";
    }

    public Book(string title, string author)
    {
        Title = title;
        Author = author;
    }

    public override string ToString()
```

```
    {
        return $"Title: {Title}, Author: {Author}";
    }
}

public class Program
{
    public static void Main()
    {
        Book b1 = new Book();
        Book b2 = new Book("C# Basics");
        Book b3 = new Book("C# Advanced", "Ali");

        Console.WriteLine(b1);
        Console.WriteLine(b2);
        Console.WriteLine(b3);
    }
}
```

- بيدي **مرونة** في إنشاء الكائنات بطرق مختلفة حسب الحاجة.
- المستخدم يقدر يختار يبني الكائن بالبيانات كلها، أو جزء منها، أو يسيبه بالـdefault.
- بيخلي الكلاس **أسهل وأوضح في الاستخدام** بدل ما يكون في method منفصلة لكل حالة.

Wafaa Mohammed - انت
الآن

🌀 إيه حكاية الـ Abstract Class في البرمجة؟

عمرك سمعت كلمة "Abstract Class" ووقفت كده تقول يعني إيه التجريد ده؟ 🤔
تعالى كده ناخدها واحدة واحدة وبأسلوب بسيط.

🤹‍♂️ تخيل معايا:

إنت داخل تشتري عربية .
أكيد أي عربية في الدنيا ليها حاجات أساسية زي:

* موتور
* عجلة دريكسيون
* فرامل

بس التفاصيل بتفرق من نوع للتاني:

* في عربية هتلاقي الموتور 1600 سي سي، والتانية 2000 سي سي.
* في عربية الفرامل عادية، والتانية فيها ABS.

هنا بييجي دور الـ **Abstract Class**.

✨ يعني إيه بقى Abstract Class؟

هو كلاس (class) بنعمله كـ"قالب عام" يحدد القواعد الأساسية لأي كلاس تاني هيورث منه.
بس خد بالك:

* الـ Abstract Class نفسه مش بنعمل منه Object (يعني مش بنادي عليه على طول).
* هو بيقول للكلاسات اللي هتورث: "أنا عندي شوية دوال (methods) لازم تنفذوها بطريقتكم الخاصة".

🛠️ مثال بسيط:

هنعمل Abstract Class اسمه **Animal** :

* فيه دالة اسمها **MakeSound()**.
* بس هو مش هيقولك الصوت إيه، هيسيبها للكلاسات اللي هتورث منه.

لما تيجي تعمل **Dog** هتنفذ MakeSound وتخليها "هوهو".
ولما تعمل **Cat** هتنفذها "مياو".

⚡ طيب إيه فايدته؟

1. يخليك منظم شغلك أكتر.
2. تضمن إن أي كلاس يورث لازم يمشي على القواعد.
3. يسهل التوسيع والتطوير بعدين.

:2

أول حاجة:

- **Coding against interface not class** = إنك تعتمد في كودك على *الواجهة (interface)* مش على *كلاس معين*.
- **Coding against abstraction not concreteness** = نفس المعنى تقريبًا، بس بشكل أوسع:
  - **Abstraction** = الحاجة العامة) زي Interface أو Abstract Class).
  - **Concreteness** = الحاجة المحددة اللي متنفذة فعلًا.(Concrete Class)

مثال:

تخيل إنت فاتح **كافيه**:

- عندك موظف اسمه."Barista"
- إنت كصاحب الكافيه مش فارق معاك الباريستا اسمه إيه أو منين.
- كل اللي يهمك إنه يلتزم بـ "قائمة مهام" زي:
  - يعمل قهوة.
  - يسخن اللبن.
  - يقدم الطلب للعميل.

هنا "قائمة المهام" دي.**Abstraction (Interface)** =
والشخص اللي بينفذها (أحمد أو محمد أو منى).**Concrete Class** =

إنت كشغلانة مش عايز تربط نفسك بشخص واحد(Concrete) ،
إنت عايز تعتمد على "المهام" اللي ممكن أي حد يعملها.(Abstraction)

في البرمجة:

لو كتبت كود بيتعامل مع:

```
PDFReport report = new PDFReport();
                   report.Generate();
```

كده إنت مربوط بـ **PDFReport بس**.

لكن لو كتبت:

```
IReport report = new PDFReport();
```

```
report.Generate();
```

دلوقتي الكود بيتعامل مع الـ **Interface (IReport).**
ممكن بكرة تغير وتخليه ExcelReport أو WordReport من غير ما تغير الكود الأساسي.

- **Coding against abstraction** = تكتب الكود معتمد على فكرة عامة.(Interface / Abstract Class)
- **Not concreteness** = من غير ما تربط نفسك بتفاصيل كلاس معين.

ده بيخلي كودك **مرن، قابل للتوسيع،** و **أسهل في الصيانة**.

:3

يعني إيه Abstraction كـ Guideline ؟

كلمة **Abstraction** معناها "التجريد" أو إنك تركز على **اللي يهمك بس** وتخفي التفاصيل اللي مش ضرورية.
كـ **Guideline** (إرشاد أو قاعدة) ← يعني وإنت بتكتب كودك، خليك دايمًا مركز على **الفكرة العامة** مش التفاصيل الصغيرة.

1. Interfaces

- بتحدد "إيه اللي لازم يتعمل" من غير ما تحدد "إزاي يتعمل."
- مثال `IShape` :فيها دالة `Draw()`.
  - المربع يرسم نفسه بشكل.
  - الدايرة ترسم نفسها بشكل تاني.

2. Abstract Classes

- نفس الفكرة، بس ممكن كمان تديلك شوية "Implementation" عام للكل، وتسيب باقي التفاصيل للـ Derived Classes.

3. Encapsulation (جزء من الـAbstraction)

- إنك تخفي التفاصيل الداخلية) زي المتغيرات (private وتسمح للمستخدم يتعامل معاك بس من خلال Methods أو Properties.

Abstraction كـ Guideline يعني:

- ما تكتبش الكود مربوط بالتفاصيل الدقيقة.
- فكر في الـ **Contract** أو الـ **واجهة عامة** اللي الناس هتتعامل معاها.
- سيب التفاصيل تتعمل في مكان تاني.(Implementations)

```
using System;
using System.Collections.Generic;

public interface IShapeSeries
{
    int CurrentShapeArea { get; set; }
    void GetNextArea();
    void ResetSeries();
}

public class SquareSeries : IShapeSeries
{
    private int side = 0;
    public int CurrentShapeArea { get; set; }

    public void GetNextArea()
    {
        side++;
        CurrentShapeArea = side * side;
    }

    public void ResetSeries()
    {
```

```
        side = 0;
        CurrentShapeArea = 0;
    }
}

public class CircleSeries : IShapeSeries
{
    private int radius = 0;
    public int CurrentShapeArea { get; set; }

    public void GetNextArea()
    {
        radius++;
        CurrentShapeArea = (int)(Math.PI * radius * radius);
    }

    public void ResetSeries()
    {
        radius = 0;
        CurrentShapeArea = 0;
    }
}
```

```
public class Shape : IComparable<Shape>
{
    public string Name { get; set; }
    public double Area { get; set; }

    public Shape(string name, double area)
    {
        Name = name;
        Area = area;
    }

    public int CompareTo(Shape other)
    {
        return this.Area.CompareTo(other.Area);
    }

    public override string ToString()
    {
        return $"{Name}, Area = {Area}";
    }
}

public abstract class GeometricShape
```

```
{
    public double Dimension1 { get; set; }
    public double Dimension2 { get; set; }

    public GeometricShape(double d1, double d2)
    {
        Dimension1 = d1;
        Dimension2 = d2;
    }

    public abstract double CalculateArea();
    public abstract double Perimeter { get; }
}

public class Triangle : GeometricShape
{
    public Triangle(double b, double h) : base(b, h) { }
    public override double CalculateArea()
    {
        return 0.5 * Dimension1 * Dimension2;
    }
    public override double Perimeter
    {
```

```
        get { return Dimension1 + Dimension2 + Math.Sqrt(Dimension1 * Dimension1 +
Dimension2 * Dimension2); }

    }

}



public class Rectangle : GeometricShape

{

    public Rectangle(double w, double h) : base(w, h) { }

    public override double CalculateArea()

    {

        return Dimension1 * Dimension2;

    }

    public override double Perimeter

    {

        get { return 2 * (Dimension1 + Dimension2); }

    }

}



public class Program

{

    public static void PrintTenShapes(IShapeSeries series)

    {

        series.ResetSeries();

        for (int i = 0; i < 10; i++)
```

```
    {
        series.GetNextArea();
        Console.WriteLine(series.CurrentShapeArea);
    }
}

public static void SelectionSort(int[] numbers)
{
    int n = numbers.Length;
    for (int i = 0; i < n - 1; i++)
    {
        int minIndex = i;
        for (int j = i + 1; j < n; j++)
        {
            if (numbers[j] < numbers[minIndex])
                minIndex = j;
        }
        int temp = numbers[i];
        numbers[i] = numbers[minIndex];
        numbers[minIndex] = temp;
    }
}
```

```
    public static void Main()
    {
        Console.WriteLine("Square Series:");
        PrintTenShapes(new SquareSeries());

        Console.WriteLine("\nCircle Series:");
        PrintTenShapes(new CircleSeries());

        Shape[] shapes = {
            new Shape("Square", 25),
            new Shape("Circle", 78.5),
            new Shape("Rectangle", 40)
        };

        Array.Sort(shapes);
        Console.WriteLine("\nSorted Shapes by Area:");
        foreach (var s in shapes)
            Console.WriteLine(s);

        Console.WriteLine("\nGeometric Shapes:");
        GeometricShape t = new Triangle(3, 4);
        GeometricShape r = new Rectangle(5, 6);
        Console.WriteLine($"Triangle Area = {t.CalculateArea()}, Perimeter =
{t.Perimeter}");
```

```
        Console.WriteLine($"Rectangle Area = {r.CalculateArea()}, Perimeter =
{r.Perimeter}");



        int[] areas = { 25, 78, 40, 10, 5 };

        SelectionSort(areas);

        Console.WriteLine("\nSorted Areas with SelectionSort:");

        foreach (var a in areas)

            Console.WriteLine(a);

    }

}
```

```
using System;


public abstract class Shape

{

    public double Dimension1 { get; set; }

    public double Dimension2 { get; set; }



    public Shape(double d1, double d2)

    {

        Dimension1 = d1;

        Dimension2 = d2;

    }



    public abstract double CalculateArea();
```

```
}

public class Rectangle : Shape
{
    public Rectangle(double w, double h) : base(w, h) { }
    public override double CalculateArea()
    {
        return Dimension1 * Dimension2;
    }
}

public class Triangle : Shape
{
    public Triangle(double b, double h) : base(b, h) { }
    public override double CalculateArea()
    {
        return 0.5 * Dimension1 * Dimension2;
    }
}

public class Circle : Shape
{
    public Circle(double r, double _) : base(r, 0) { }
```

```
    public override double CalculateArea()
    {
        return Math.PI * Dimension1 * Dimension1;
    }
}

public class ShapeFactory
{
    public Shape CreateShape(string shapeType, double dim1, double dim2)
    {
        switch (shapeType.ToLower())
        {
            case "rectangle":
                return new Rectangle(dim1, dim2);
            case "triangle":
                return new Triangle(dim1, dim2);
            case "circle":
                return new Circle(dim1, 0);
            default:
                throw new ArgumentException("Invalid shape type");
        }
    }
}
```

```
public class Program
{
    public static void Main()
    {
        ShapeFactory factory = new ShapeFactory();

        Shape rect = factory.CreateShape("rectangle", 5, 6);
        Shape tri = factory.CreateShape("triangle", 4, 3);
        Shape circ = factory.CreateShape("circle", 7, 0);

        Console.WriteLine($"Rectangle Area = {rect.CalculateArea()}");
        Console.WriteLine($"Triangle Area = {tri.CalculateArea()}");
        Console.WriteLine($"Circle Area = {circ.CalculateArea()}");
    }
}
```

.