Interfaces

LCSCI5202: Object Oriented Design Week 5

What is an Interface

- An interface is a contract that defines a set of methods and properties without implementing them. Any class or struct that implements an interface agrees to implement those members.
- Interfaces are declared using the interface keyword.

```
public interface IVehicle {
    void StartEngine();
    void StopEngine();
    int CurrentSpeed { get; }
}
```

Key Characteristics

- No implementation: Interfaces contain only method and property declarations, no code.
- What Interface can’t contain: Fields (variables), Constructors, Any implementation code
- Multiple implementation: A class can implement multiple interfaces, unlike inheritance from classes.
- Interface members are always public.
- Conventionally, interface name starts with ”I”; such as **IComparable, Ienumerable.**
- Every method and property from interface must be implemented in Class.
- Method names, return types, and parameters must match exactly in Class.
- Interface implementations must be publicly accessible.

Example

```
public interface IVehicle
{
void StartEngine();
void StopEngine();
int CurrentSpeed { get; }
}

public class Car : IVehicle
{

public int CurrentSpeed { get; private set; }

public void StartEngine()
{
Console.WriteLine("Car engine started");
CurrentSpeed = 10;
}

public void StopEngine()
{
Console.WriteLine("Car engine stopped");
CurrentSpeed = 0;
}

public void PlayMusic()
{
Console.WriteLine("Playing music...");
}

}
```

```
static void Main(string[] args)
{

Car car2 = new Car();

IVehicle vehicle = new Car();

vehicle.StartEngine();

Console.WriteLine(vehicle.CurrentSpeed);

vehicle.StopEngine();

car2.PlayMusic();

}
```

Polymorphism in Interfaces

- Interface variables can hold any implementing class
  - A variable of type IVehicle can point to a Car, Truck, or Motorcycle object
- Same method call, different behavior
  - Calling vehicle.StartEngine() executes different code depending on the actual object type
- Code works with the interface contract, not specific classes
  - Your methods only need to know about IVehicle, not Car, Truck, or Motorcycle
- Add new types without changing existing code
  - Create a new Helicopter class implementing IVehicle - all existing methods automatically work with it

Polymorphism in Interfaces

Without Interface Polymorphism

```
public void DriveCar(Car car) {
car.StartEngine();
Console.WriteLine("Driving the car");
}

public void DriveTruck(Truck truck) {
truck.StartEngine();
}

public void DriveMC(Motorcycle bike) {
bike.StartEngine();
}
```

Polymorphism (With Interface Polymorphism - One Method)

```
public interface IVehicle {

void StartEngine();

string GetVehicleType();

}

public class Car : IVehicle {

public void StartEngine(){ Console.WriteLine("Car: Vrroom!");
}

public string GetVehicleType(){
return "Car"
}

}

public class Truck : IVehicle {

public void StartEngine(){ Console.WriteLine("Truck: VRROOOM!");
}

public string GetVehicleType()
{
return "Truck";
}
```

```
public void TestDrive(IVehicle vehicle) {
Console.WriteLine($"Test driving:
{vehicle.GetVehicleType()}");
vehicle.StartEngine();
}
static void Main(){

TestDrive(new Car());
TestDrive(new Truck());
TestDrive(new Motorcycle());
}
```

Interface Collections - Store Different Types Together

```
List<IVehicle> fleet = new List<IVehicle>();
fleet.Add(new Car());
fleet.Add(new Truck());
fleet.Add(new Motorcycle());

foreach (IVehicle vehicle in fleet) {
        vehicle.StartEngine();
}
```

Multiple interfaces

```
public interface IVehicle {
void StartEngine();
}
public interface IFlyable {
        void Fly();
        }
public class FlyingCar : IVehicle, IFlyable {
public void StartEngine() {
        Console.WriteLine("Engine started");
}
public void Fly() {
Console.WriteLine("Taking off!");
}
}
FlyingCar cool = new FlyingCar();
cool.StartEngine();
cool.Fly();
```

Interface Inheritance

```
public interface IShape {
double GetArea();
}
public interface IColoredShape : IShape {
string Color { get; set; }
}
public class ColoredCircle : IColoredShape {
        public double Radius { get; set; }
        public string Color { get; set; }
        public double GetArea() {
return Math.PI * Radius * Radius;
}
}
ColoredCircle circle = new ColoredCircle();
circle.Radius = 5;
circle.Color = "Red";
Console.WriteLine($"Area: {circle.GetArea()}, Color: {circle.Color}");
```

When to use interfaces

- Designing for flexibility: Use interfaces when you need to design systems that are expected to evolve or have interchangeable parts.
- Defining common behaviors: When different classes should share a common set of methods but don’t share a base class.
- Dependency Injection: Interfaces are commonly used in dependency injection to swap implementations without changing client code.

When Interfaces Are Overkill

- Only One Implementation

```
public interface IUser {
string Name { get; set; }
string Email { get; set; }
}


public class User : IUser {
public string Name { get; set; }
public string Email { get; set; }
}

// Better Option
public class User {
public string Name { get; set; }
public string Email { get; set; }
}
```

Interface Best Practices

Do:

- Keep interfaces small (3-5 methods max)
- Name with "I" prefix (IPlayable)
- Think about flexibility
- Document what methods should do

Don’t:

- Create interfaces for everything
- Make "fat" interfaces with 10+ methods
- Forget "public" on implementations
- Use vague names (IManager, IHelper)

Summary

- Define a contract of methods/properties a class must implement.
- Contain no implementation, only declarations.
- Classes can implement multiple interfaces.
- Enable polymorphism — same code works for many types.
- Interfaces can inherit other interfaces.
- Use when different classes share common behavior.
- Avoid when there’s only one implementation.
- Best practice: keep them small, clear, and purpose-driven.

Activity

Create interfaces and classes for:

- IAttackable - has Attack() method and AttackPower property
- IDefendable - has Defend() method and DefensePower property
- Warrior class - implements both interfaces
- Mage class - implements IAttackable only
- Battle method - accepts any IAttackable

Create a Main Function and call the methods and class objects