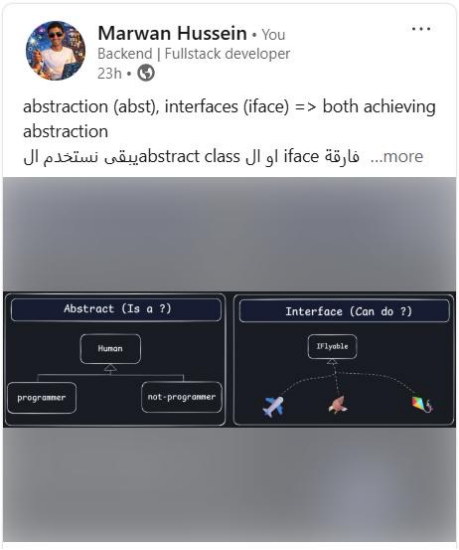
Part01

1. Why is it better to code against an interface rather than a concrete class?
   - Loose coupling -> a code depends on a contract not a specific implementation
   - Allow polymorphism
   - Testability
2. When should you prefer an abstract class over an interface?
   When we want to provide shared implementation (for common base behaviour)
3. How does implementing IComparable improve flexibility in sorting?
   Allow objects to implement their own oredering
   And the sorting algorithm remains stable with the same complexity
4. What is the primary purpose of a copy constructor in C#?
   Deep copy: coping obj's data not the address directly
5. How does explicit interface implementation help in resolving naming conflicts?
   Regular naming: PascalCase Interfaces naming: IPascalCase
6. What is the key difference between encapsulation in structs and classes?
   Encapsulation in structs ensures each copy has its own independent data, while in classes multiple references can point to the same object
7. what is abstraction as a guideline, what's its relationship with encapsulation?
   Hiding implementation details and exposing only essential features (e.g. IVehicle.StartEngine()) without showing how it works.
8. How do default interface implementations affect backward compatibility in C#?
   Allowing to add new methods to interfaces without breaking existing implementations.
   Classes that don't have override the new method -> automatically inherit the default behavior.
9. How does constructor overloading improve class usability?
   Flexibility in obj creation (initialize full/partial/default data)
   Readability: for developers using the class.

Part02

1. LinkedIn article about abstract class ?



2. What we mean by coding against interface rather than class? and if u get it so What we mean by code against abstraction not concreteness?
   a. Coding against a class ur code depends directly on a concrete implementation.

```
Plane p = new Plane();
p.FlyByMotor();
Bird b = new Bird();
b.FlyByWings();
```

   Coding against an Iface: ur code depends on a contract rather than a specific implementation

```
IFlyable p = new Plane();
p.Fly();
IFlyable b = new Bird();
b.Fly()
```

   b. Coding against abstraction not concreteness: designing to rely on general behaviors rather than specific implementation

```
Human p = new Programmer();
p.Breathe();
// p.Sleep(); Error -> Programmer cant sleep
Human notP = new NotProgrammer();
notP.Breathe();
notP.Sleep();
```

3. What is abstraction as a guideline and how we can implement this through what we have studied?

   Abstraction: is a principle to hide complexity and expose only the essential features

```
internal abstract class Animal
{
    public abstract void MakeSound();
    public abstract void Walk();
    public void Breathe() { BreatheFromLungs(); }
    public void Eat() { EatFromMouth(); }
}
```

```
internal class Dog() : Animal
{
    public override void MakeSound() {Bark();}
    public override void Walk() { Use4Feets(); }
}
```

```
internal class Snake() : Animal
{
    public override void MakeSound() { Hiss(); }
    public override void Walk() { UseBody(); }
}
```

Part03 (Bonus)

What is operator overloading?

To overload operators like (<,>,==,!=, +,-,*,/ ..etc) on some class based on what I want => like point1 + point2 // error

Else if we defined an operator overload to define how to add point1 to point2 for example