Question01:

Primarily because it promotes a set of principles that enhance the maintainability, flexibility, and testability of a system and it promotes several key software design principles, primarily loose coupling. This approach is a core tenet of the Dependency Inversion Principle.

Question02:

Shared Implementation: When you want to provide a common, implemented behavior that all or most subclasses should inherit, reducing code duplication.

Question03:

Implementing IComparable improves flexibility by providing a default sorting behavior that works automatically with .NET's built-in features.

Question04:

Is to create a new object as a precise copy of an existing instance of the same class, ensuring that the new object starts with the same data.

Question05:

Explicit interface implementation allows a class to implement interface methods separately from its own methods, avoiding naming conflicts and enabling different behaviors depending on how the object is referenced.

Question06:

Encapsulation works the same in both, but structs are value types copied by value while classes are reference types copied by reference.

Question07:

Encapsulation hides variables or some implementation that may be changed so often in a class to prevent outsiders access it directly. They must access it via getter and setter methods, while abstraction is used to hide something too, but in a higher degree (class, interface). Clients who use an abstract class (or interface) do not care about what it was, they just need to know what it can do.

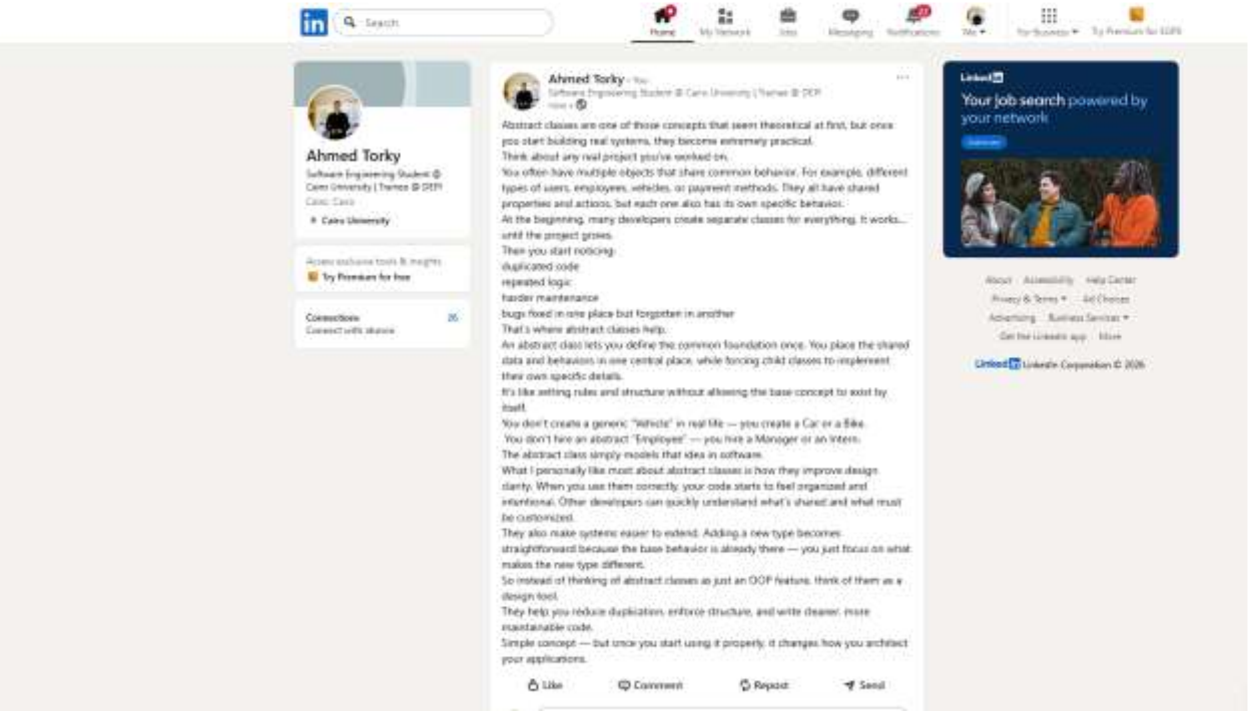
Question08:

primarily improve backward compatibility by allowing API authors to add new methods to an interface without breaking existing classes that implement it.

Question09:

Constructor overloading improves class usability by providing flexibility, convenience, and cleaner code when initializing objects. It allows a class to have multiple constructors with different parameter lists, enabling developers to instantiate objects in various ways based on the available data.

Part02:



Ahmed Torky • You
Software Engineering Student @ Cairo University | Trainee @ DEPI

Abstract classes are one of those concepts that seem theoretical at first, but once you start building real systems, they become extremely practical.
Think about any real project you've worked on.
You often have multiple objects that share common behavior. For example, different types of users, employees, vehicles, or payment methods. They all have shared properties and actions, but each one also has its own specific behavior.
At the beginning, many developers create separate classes for everything. It works… until the project grows.
Then you start noticing:
duplicated code
repeated logic
harder maintenance
bugs fixed in one place but forgotten in another
That's where abstract classes help.
An abstract class lets you define the common foundation once. You place the shared data and behaviors in one central place, while forcing child classes to implement their own specific details.
It's like setting rules and structure without allowing the base concept to exist by itself.
You don't create a generic "Vehicle" in real life — you create a Car or a Bike.
You don't hire an abstract "Employee" — you hire a Manager or an Intern.
The abstract class simply models that idea in software.
What I personally like most about abstract classes is how they improve design clarity. When you use them correctly, your code starts to feel organized and intentional. Other developers can quickly understand what's shared and what must be customized.
They also make systems easier to extend. Adding a new type becomes straightforward because the base behavior is already there — you just focus on what makes the new type different.
So instead of thinking of abstract classes as just an OOP feature, think of them as a design tool.
They help you reduce duplication, enforce structure, and write cleaner, more maintainable code.
Simple concept — but once you start using it properly, it changes how you architect your applications.

Question02:

Coding against an interface means depending on the interface instead of a specific concrete class. The interface defines what operations are available, not how they are implemented. This allows different classes to implement the same interface and be used interchangeably.

Coding against abstraction means depending on abstract types such as interfaces or abstract classes instead of concrete implementations. Concrete classes represent details that may change, while abstractions represent stable behavior.

Question03:

Abstraction is the process of hiding implementation details and exposing only essential behavior. It reduces complexity and improves flexibility.

abstraction is implemented using:

1. interfaces
2. abstract classes
3. encapsulation

These tools help us design systems that are loosely coupled, maintainable, and easy to extend.

Part03:

Question02:

Operator overloading in C# is a feature that allows developers to redefine the behavior of built-in operators (like +, -, *, ==) for classes or structs. This provides a custom implementation for an operation when one or both operands are of a custom type, making the code more intuitive and readable, especially for mathematical or domain-specific objects like complex numbers, vectors, or currency.