Question1:

In C#, the compiler automatically provides a parameterless default constructor if you don’t define any constructors yourself.

- This auto-generated constructor simply initializes fields to their default values (e.g., 0 for numbers, null for reference types).
- However, once you define any custom constructor, the compiler assumes you want full control over how objects are initialized.
- To avoid ambiguity, the compiler stops generating the default constructor automatically.

Question2:

Method overloading means defining multiple methods with the same name but different parameter lists. In C#, this allows you to perform similar operations with different inputs while keeping the method name consistent.

- Improves readability
  Instead of having separate methods like SumTwoIntegers, SumThreeIntegers, and SumTwoDoubles, you can just call Sum() with the appropriate arguments. This makes the code easier to

read and understand because the intent is clear — you're summing values, regardless of type or count.

- Enhances reusability
  You don't need to invent new method names for every variation. The same method name can handle different scenarios, so developers can reuse the same logical operation across different data types or argument counts. This reduces duplication and makes maintenance easier.

Question3:

Constructor chaining in inheritance ensures that when you create an object of a derived class, the base class constructor runs first. This guarantees that all inherited properties are properly initialized before the derived class adds its own logic.

- Orderly initialization: The base class sets up its part of the object before the child class adds new properties.
- Code reuse: Instead of duplicating initialization logic in the child, you reuse the base constructor.
- Consistency: It enforces a predictable flow of object creation, reducing bugs and confusion.

Question4:

new keyword (method hiding)

- When you use new, the derived class method *hides* the base class method.
- The base method still exists, but if you reference the object through a base class variable, the base method will be called.
- It's essentially saying: *"I know there's a method in the base class with the same name, but I want to define a new one that hides it."*

- override keyword (true overriding)
- When you use override, the derived class method *replaces* the base class's virtual method.
- Even if you reference the object through a base class variable, the derived class's overridden method will be called.

Question5:

By default, the ToString() method in C# returns the class name (e.g., "Namespace.ClassName"), which isn't very useful when you want to see the actual data inside an

object. Overriding ToString() makes your objects more readable and meaningful.

- Improves readability: Instead of generic text, you get a clear representation of the object's state (like (X, Y, Z) instead of Child).
- Useful for debugging: When printing objects in logs or the console, overridden ToString() shows property values directly, saving time.
- Better user experience: In UI or reports, objects display in a human-friendly format rather than technical identifiers.

Question6:

An interface only defines *what* methods and properties a class must have, but not *how* they work.

- It has no implementation on its own, so there's nothing to instantiate.
- You can only create objects of classes that implement the interface, because those classes provide the actual logic.

Question7:

Backward compatibility: You can add new methods to interfaces without breaking existing implementations.

- Reduced boilerplate: Classes don't need to implement common logic repeatedly; they can rely on the default.
- Flexibility: Classes can override the default if they need custom behavior, but otherwise reuse the provided one.
- Cleaner design: Interfaces can evolve over time without forcing all implementers to change immediately.

Question8:

Flexibility: You can write code that works with any class implementing the interface, not just one specific class.

- Polymorphism: The same interface reference can point to different objects (e.g., Car, Bike, Robot), and calling Move() will run the correct implementation.
- Decoupling: Your code depends on the interface, not the concrete class, making it easier to extend or modify later.
- Maintainability: If you add new classes that implement the interface, existing code doesn't need to change — it just works.

Question9:

In C#, a class can inherit from only one base class (single inheritance).

- However, a class can implement multiple interfaces, which allows it to inherit behavior contracts from many sources.
- This provides the flexibility of multiple inheritance without the complexity and ambiguity (like the “diamond problem” in C++).
- Interfaces act as contracts, ensuring that classes implement specific methods, while still keeping the design clean and avoiding conflicts.

Question10:

Here’s the answer in plain words, without a table:

A virtual method in C# is a method that already has an implementation in the base class, but it can be optionally overridden in a derived class if you want to change its behavior. It gives you a default version that you can either use as-is or customize.

An abstract method, on the other hand, has no implementation at all in the base class. It’s just a declaration that says: “Any derived class must provide its

own version of this method.” Because of that, abstract methods can only exist inside abstract classes, and overriding them in derived classes is mandatory.

Part2:

1- Difference between **class** and **struct** in C#

- **Class**:
  - Reference type (stored on the heap).
  - Supports inheritance.
  - Can have a destructor.
  - Variables can be null.
  - When passed around, you’re passing a reference (changes affect the original object).
- **Struct**:
- Value type (stored on the stack).
- Does **not** support inheritance (but can implement interfaces).

- No destructor.
- Cannot be null unless declared as Nullable.
- When passed around, you're passing a copy (changes don't affect the original).

2- Other relations between classes (besides inheritance)

Inheritance is just one way classes relate. Other important relationships include:

- **Association**: A general connection between two classes (e.g., a Teacher teaches a Student).
- **Aggregation**: A "has-a" relationship where one class contains another, but the contained object can exist independently (e.g., a Library has Books).
- **Composition**: A stronger "has-a" relationship where the contained object's lifecycle depends on the container (e.g., a Car has an Engine — if the car is destroyed, the engine goes too).
- **Dependency**: One class depends on another temporarily (e.g., a Printer depends on a Document to print).
- **Polymorphism**: Different classes can be treated as the same type if they share a base class or interface.