

C# Day 7

Question 1: Custom Constructor and Default Constructor

Why does defining a custom constructor suppress the default constructor in C#?

When you define any custom constructor, the compiler no longer provides a parameterless default constructor automatically. This is by design to ensure explicit initialization—if you’ve defined specific ways to construct objects, the compiler assumes you don’t want a no-argument constructor unless you explicitly define one.

Question 2: Method Overloading Benefits

How does method overloading improve code readability and reusability?

Method overloading allows using the same method name for related operations with different parameter types or counts. This improves readability by providing a consistent interface (e.g., `Sum()` for all addition operations) and reusability by eliminating the need for multiple method names like `SumIntegers()`, `SumDoubles()`, etc.

Question 3: Constructor Chaining Purpose

What is the purpose of constructor chaining in inheritance?

Constructor chaining ensures that the base class is properly initialized before the derived class. Using the `base()` keyword, the derived class can invoke the parent constructor, avoiding code duplication and maintaining proper initialization order in the inheritance hierarchy.

Question 4: new vs override Keywords

How does new differ from override in method overriding?

- **override:** Provides true polymorphism; the derived class method is called even when using a base class reference.
- **new:** Hides the base class method; the method called depends on the reference type, not the actual object type. This breaks polymorphism.

Question 5: ToString() Override Purpose

Why is ToString() often overridden in custom classes?

The default ToString() returns the fully qualified type name, which is not useful. Overriding it provides a meaningful string representation of the object's state, making debugging easier and improving output readability when printing objects.

Question 6: Interface Instantiation

Why can't you create an instance of an interface directly?

Interfaces define contracts (method signatures) without implementation. They cannot be instantiated because they lack concrete behavior. You can only create instances of classes that implement the interface and provide the actual method implementations.

Question 7: Default Interface Implementations

What are the benefits of default implementations in interfaces introduced in C# 8.0?

Default interface implementations allow adding new methods to interfaces without breaking existing implementations. Classes implementing the interface automatically inherit the default behavior unless they choose to override it, enabling interface evolution and reducing code duplication.

Question 8: Interface References

Why is it useful to use an interface reference to access implementing class methods?

Using interface references enables polymorphism and loose coupling. Code depends on abstractions (interfaces) rather than concrete implementations, making it more flexible, testable, and maintainable. Different classes can be substituted seamlessly as long as they implement the interface.

Question 9: Single Inheritance Limitation

How does C# overcome the limitation of single inheritance with interfaces?

While C# allows a class to inherit from only one base class (single inheritance), it permits implementing multiple interfaces. This provides the benefits of multiple inheritance (inheriting contracts from multiple sources) while avoiding the complexity and ambiguity issues associated with multiple class inheritance.

Question 10: Virtual vs Abstract Methods

What is the difference between a virtual method and an abstract method in C#?

- **Virtual method:** Has a default implementation in the base class; derived classes can optionally override it.
- **Abstract method:** Has no implementation in the base class; derived classes must override and implement it. Abstract methods can only exist in abstract classes.

Part 2 - Question 1: Class vs Struct

What is the difference between class and struct in C#?

- **Class:** Reference type stored on the heap; supports inheritance; passed by reference; can be null.
- **Struct:** Value type stored on the stack; no inheritance (except from interfaces); passed by value; cannot be null (unless nullable struct).

Structs are suitable for small, lightweight data structures, while classes are used for complex objects and OOP designs.

Part 2 - Question 2: Class Relationships

If inheritance is a relation between classes, clarify other relations between classes.

- **Inheritance (Is-A):** A derived class is a specialized version of the base class (e.g., Car is a Vehicle).
- **Association (Has-A):** One class uses or interacts with another (e.g., Teacher teaches Student).
- **Aggregation:** Weak "has-a" relationship; contained objects can exist independently (e.g., Department has Employees).
- **Composition:** Strong "has-a" relationship; contained objects cannot exist without the container (e.g., House has Rooms).
- **Dependency:** One class depends on another temporarily (e.g., method parameter usage).