Part01

1 **Why can't you override a sealed property or method?**

A sealed method or property is explicitly marked to stop further overriding.
Once sealed, it guarantees a stable implementation that cannot be changed in derived classes. This prevents unexpected behavior and enforces design constraints.

---

2 **What is the key difference between static and object members?**

- **Static members belong to the class itself, not to any specific object. They can be accessed without creating an instance.**
- **Object (instance) members belong to each individual object and require an instance to be accessed.**

---

3 **Can you overload all operators in C#?**

No. Not all operators can be overloaded.
Only a subset of operators (+, -, *, /, %, ==, !=, <, >, <=, >=, etc.) can be overloaded.
Operators like . (member access), ?: (ternary), = (assignment) cannot be overloaded because they are fundamental to the language's behavior.

---

4 **When should you consider changing the underlying type of an enum?**

You should change the underlying type of an enum when:

- **You need to save memory (e.g., use byte instead of int).**
- **You want the enum to match data stored in a database or external API with a specific numeric type.**
- **You are working with interop scenarios where the enum must match a particular type.**

---

5 **Why can't a static class have instance constructors?**

A static class cannot be instantiated. Since constructors are only used to create instances, having an instance constructor in a static class would not make sense. Static classes can only have a static constructor, which initializes static members once before first use.

---

6 **What are the advantages of using Enum.TryParse over direct parsing with int.Parse?**

- **Enum.TryParse directly maps a string to a defined enum value safely.**
- **It prevents exceptions by returning false if parsing fails.**
- **It makes code more readable and avoids manual conversion from int to enum.**
- **It supports case-insensitive parsing if specified.**

---

7 **What is the difference between overriding Equals and == for object comparison in C# struct and class?**

- **Equals: Can be overridden in both classes and structs to define value-based equality.**
- **==: By default, compares references for classes and is not defined for structs unless overloaded.**
  If you want == to mean value equality for structs or classes, you must overload it explicitly.

---

8 **Why is overriding ToString beneficial when working with custom classes?**

Overriding ToString provides a human-readable string representation of an object.
It helps with debugging, logging, and displaying meaningful information instead of just the class name.

9️⃣ Can generics be constrained to specific types in C#? Provide an example.

Yes. Generics can use constraints to limit which types are allowed.

Example:

```
public class Repository<T> where T : IEntity
{
  public void Save(T entity) { /* ... */ }
}
```

Here, T must implement the IEntity interface.

🔟 What are the key differences between generic methods and generic classes?

- **Generic methods: Apply generics only to a specific method. The rest of the class may not be generic.**
- **Generic classes: The entire class is generic, and all its members can work with the type parameter.**

1️⃣1️⃣ Why might using a generic swap method be preferable to implementing custom methods for each type?

A generic swap method works with any type.
It avoids code duplication and provides flexibility, so you don't have to write separate swap methods for int, string, Rectangle, etc.

1️⃣2️⃣ How can overriding Equals for the Department class improve the accuracy of searches?

If Equals is overridden, two Department objects with the same logical value (e.g., both named "HR") will be treated as equal.
This ensures that searches, comparisons, and collections (like dictionaries or hash sets) work correctly, even if the objects are different instances.

1️⃣3️⃣ Why is == not implemented by default for structs?

- **Structs inherit ValueType.Equals, which already provides value comparison.**
- **Implementing == by default could be inefficient or ambiguous for large structs.**
- **C# leaves it to the developer to explicitly define what == means for their struct by overloading it.**

Part02+Bonus

1️⃣ What do we mean by Generalization concept using Generics?

Generics in C# are a way to write **general-purpose, reusable code** without tying it to a specific data type.

- Instead of writing separate methods or classes for int, string, double, etc., you use a **type parameter (T)**.
- This is called **generalization** because the logic is written once but can work with any type safely.

Example:

```
public class Box<T>
{
   public T Value { get; set; }
}
```

Here, Box<int> and Box<string> both use the same logic but with different data types.

➡️ Generalization = Avoid duplication + Strong type safety.

2️⃣ What do we mean by Hierarchy design in real business?

Hierarchy design means organizing classes, objects, or roles in a **parent–child (inheritance) structure**, similar to how businesses are structured.

- In business, you may have:
    - **CEO → Manager → Employee** (real-world hierarchy).
- In programming, you design classes the same way:
    - **Employee (base class) → Manager (derived class) → CEO (derived from Manager)**.

This helps to **reuse code**, enforce rules, and model real-world relationships inside software systems.

➡️ Hierarchy design = mapping real business roles/entities into inheritance or role-based class structures.

3️⃣ What is Event-driven programming?

Event-driven programming is a style of programming where the **flow of the program is controlled by events** (user actions or system signals).

- Events: clicks, key presses, mouse movements, timer ticks, data received from network, etc.
- The program responds to these events using **event handlers** (methods that run when the event occurs).

Example in C#:

```
button.Click += (sender, e) => Console.WriteLine("Button was clicked!");
```

➡️ Event-driven programming is widely used in **GUIs, games, and web applications**, where the program waits for the user's actions and reacts accordingly.