Value type vs Reference Type:

A **value type** is a data type in computer programming that directly stores the data's value, rather than a reference to the data's location in memory. Value types are typically small, simple data types, and they are characterized by the following properties:

**Immutability**: Value types are often immutable, meaning their values cannot be changed once they are created. Any operation that appears to modify a value type creates a new instance with the modified value.

**Value Semantics**: Equality comparisons for value types are based on their actual values rather than memory references. Two instances of the same value type with identical values are considered equal.

**Stack Allocation**: In some programming languages, value types are allocated on the stack rather than the heap. This can lead to efficient memory management.

Examples of value types include integers, floating-point numbers, characters, and structs in C#, among others.

**Reference Type**:

A **reference type** is a data type in computer programming that stores a reference or pointer to the memory location where the actual data is stored. Reference types are typically used for more complex, large, or mutable data structures and are characterized by the following properties:

**Reference Semantics**: Equality comparisons for reference types are based on memory references, not the actual values. Two reference variables are considered equal if they point to the same memory location.

**Heap Allocation**: Reference types are often allocated on the heap, which allows for more dynamic memory management and the ability to share data among multiple parts of a program.

**Mutability**: Reference types can be mutable, meaning their data can be modified directly through references.

Examples of reference types include classes, objects, arrays, and custom user-defined classes in many programming languages.

2. Class vs Struct

In basic terms, a **struct** is a value type while a **class** is a reference type. Value types contain their data directly on the stack, while reference types store a reference to an object containing the data on the heap.

Structs are often used to represent simple data types, such as **integers**, **strings**, and other basic data types. Classes, on the other hand, are used to represent more complex objects with **multiple properties and methods**.

Another important difference between structs and classes is that structs are value types, which means that they are **copied** when they are passed as arguments to methods or functions.

Due to their memory allocation differences, structs are generally **faster** than classes. If you’re working with a large amount of data, structs can be more efficient because they don’t require the overhead of heap memory allocation.

However, there are some cases where classes are faster than structs. For example, when copying large objects, classes can be more efficient because they only copy a reference to the object instead of the object itself.

One common mistake when using structs is to make them too large. If your struct is too large, it will be stored on the heap instead of the stack, which can cause **performance issues**.

**3. Scoped vs Transient vs Singleton Service in C#**

Explaining the Scoped Service in C#

Have you ever felt like you’ve entered a unique space only to find out it already exists in another context? That’s how a scoped service behaves!

Imagine your application represents a hotel building. Your request is to get a hotel room – that’s your “scope”. Each scoped service is unique per request, like each hotel room being unique for each guest. A new instance is created for each scope. It creates one instance per request. (jotobar e use hok akta instance theke serve korbe)

*// Registering a Scoped Service*

services.AddScoped<IMyService, MyService>();

With this code, we register a service as scoped. Whenever a new request is made by the application, a new instance is created. Different requests? Different instances!

The beauty of Scoped service is in its ability to share resource within a request. Time-save? Heck, yes!

However, be warned: misusing scoped services can result in cross-request contamination and concurrency issues. Always handle with care!

Explaining the Transient Services in C#

Transient services are like butterflies, a different one with each glance! They create a new instance each time they’re requested, providing all classes with a unique copy. Let’s take a look at the registration code. It can create multiple instance per request.(jotobar lagbe toto ta instance create korbe)

*// Registering a Transient Service*

services.AddTransient<IMyService, MyService>();

Each call to GetService<IMyService> gets a fresh instance like receiving a new gadget each time you ask!

Transient services lend themselves to lightweight, stateless services yet careful consideration is required. Overuse can lead to increased memory usage. Note to self: More isn’t always merry!

**Explaining the Singleton Service in C#**

One ring to rule them all, one ring to bind them! If services were rings, Singleton would be the One ring. It creates a single instance for the whole application. Registered once, used everywhere!

*// Registering a Singleton Service*

services.AddSingleton<IMyService, MyService>();

Order up a Singleton service, and you’re provided an instance. Make a second request and you’ll get the same instance. It’s persistent!

Singleton services are great for sharing states across requests and even connections. Exercise caution though, singletons don’t come without tripwires. Remember, because they hang around for longer than scoped or transient, you might find them hoarding resources. It creates one instance in one life cycle.

**Scoped Services**

Think of scoped services as your personal assistant during a trip. It’s there when you embark, serves all your needs during journey, and concludes its role once the journey ends. It retains information through a specific scope/request, but this doesn’t persist across different scopes.

Scoped services are beneficial when multiple objects in a single scope need to communicate or share data.

**Transient Services**

Transient services, in contrast, are more like hitchhikers on separate journeys. They join you momentarily before disembarking to join another journey. They don’t retain any memory or state from previous journeys.

If you need a service in separate classes which doesn’t retain data and doesn’t need to communicate with each other, transient services are your answer.

**Singleton Services**

Singleton services are your loyal companions. Initiated once, they stick around throughout the application lifetime.

Singletons are useful if you need to maintain stateful information that persists through different requests in the same application session.

**When to Use Each Service**

**Scoped Service**: Best suited when you want to maintain state within a single request, but not persistently. This is also ideal when you need shared communication/data-access within object instances of a single request.

**Transient Service**: Optimal choice for lightweight, stateless services that are implemented throughout your application without needing integration or communication. These instances do not remember their previous state – like state amnesia!

**Singleton Service**: Preferable for dealing with data or state that needs to be shared across multiple requests, or when the instantiation process is expensive. Keep an eye on state management and thread-safety when using singleton services.

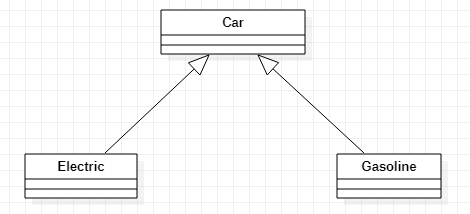
4. **Why no multiple inheritance**

C# does not support multiple inheritance of classes, meaning a class cannot directly inherit from more than one class at the same time. This limitation was introduced for several reasons, including reducing complexity, avoiding the "diamond problem" (a naming conflict that arises with multiple inheritance), and promoting better code organization through interfaces.

# 5. What is a diamond problem in Object-Oriented Programming?

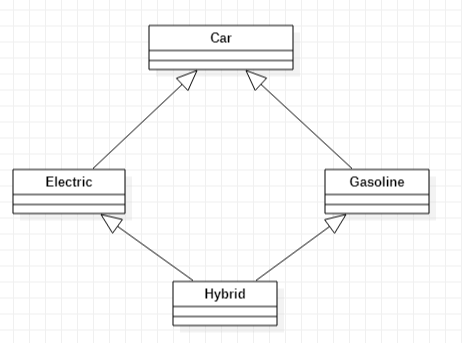
## Example

Consider the example of a car.



Both electric and gasoline cars inherit the properties of a car.

Now, there is a special case if there is another class – a Hybrid class, for example – that inherits both the Electric and Gasoline class. We can see in the diagram that it will form a diamond.



The hybrid car is both an electric car and a gasoline car. These kinds of special cases will result in a **diamond problem**.

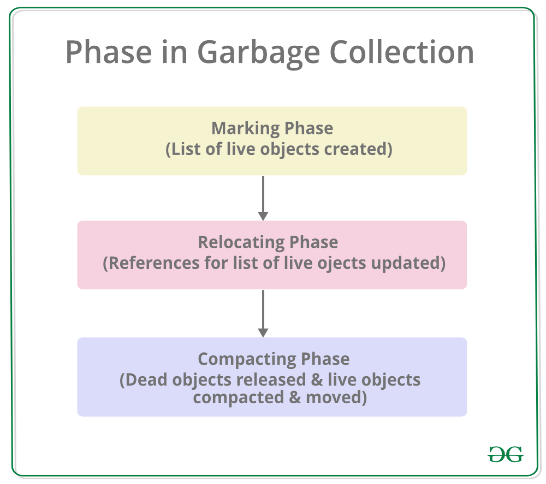
This diamond creates a problem, because now the Hybrid class has two copies of the Car class for each path.

6. **Garbage Collection in c#:**

Garbage collection is a memory management technique used in the .NET Framework and many other programming languages. In C#, the garbage collector is responsible for managing memory and automatically freeing up memory that is no longer being used by the application.

The garbage collector works by periodically scanning the application’s memory to determine which objects are still being used and which are no longer needed. Objects that are no longer being used are marked for garbage collection, and their memory is freed up automatically by the garbage collector.

There are mainly **3** phases in garbage collection. Details about these are given as follows:



𝐈𝐄𝐧𝐮𝐦𝐞𝐫𝐚𝐛𝐥𝐞 vs 𝐈𝐐𝐮𝐞𝐫𝐲𝐚𝐛𝐥𝐞

𝐈𝐄𝐧𝐮𝐦𝐞𝐫𝐚𝐛𝐥𝐞 serves as a fundamental interface suitable for in-memory collections, offering lazy loading and executing queries in-memory through LINQ extension methods. Conversely, 𝐈𝐐𝐮𝐞𝐫𝐲𝐚𝐛𝐥𝐞 is a generic interface that extends 𝐈𝐄𝐧𝐮𝐦𝐞𝐫𝐚𝐛𝐥𝐞 and is tailored for querying external data sources. It supports deferred execution, translating queries into a format understandable by the underlying data source, such as a database, thereby optimizing performance for large datasets.

The key differences lie in their execution strategies, where 𝐈𝐄𝐧𝐮𝐦𝐞𝐫𝐚𝐛𝐥𝐞 processes queries in-memory, and 𝐈𝐐𝐮𝐞𝐫𝐲𝐚𝐛𝐥𝐞 defers execution and performs server-side processing, making it more suitable for scenarios involving significant data volumes and external data sources. The choice between them hinges on whether the data resides in-memory or originates from an external source, influencing the trade-offs between flexibility and performance.