Questions Asked in Various Interviews

**What is SQL Injection?**

SQL injection is a type of cyberattack that targets applications and websites with vulnerabilities in their database-related code. It occurs when an attacker manipulates input data in such a way that it is treated as SQL (Structured Query Language) code by the application's database. This can allow the attacker to execute arbitrary SQL queries on the database, potentially gaining unauthorized access to, modifying, or deleting data, and even taking control of the application itself.

SQL injection typically happens when an application does not properly validate or sanitize user inputs before including them in SQL queries. This allows attackers to insert malicious SQL statements into the input fields of a web form or URL parameters, which are then executed by the application's database. Common targets for SQL injection include login forms, search fields, and any other input mechanisms that interact with a database.

The consequences of a successful SQL injection attack can be severe, including:

**1. Unauthorized data access:** Attackers can extract sensitive data from the database, such as user credentials, financial records, or personal information.

**2. Data modification:** Attackers can alter, delete, or insert data into the database, potentially causing data corruption and integrity issues.

**3. Application compromise:** Attackers may gain control over the application or underlying server, enabling them to execute malicious commands or upload malware.

To prevent SQL injection attacks, developers should follow best practices like using prepared statements and parameterized queries, validating and sanitizing user inputs, and employing security mechanisms like Web Application Firewalls (WAFs) to filter out malicious requests. Regular security testing and audits of applications can help identify and fix vulnerabilities before they are exploited by attackers.

**What is Memory Leakage and how to avoid it?**

Memory leakage, also known as a memory leak, is a common software issue that occurs when a program or application does not properly release memory that it has allocated and no longer needs. Over time, these unreleased blocks of memory accumulate, consuming system resources and potentially causing the application or system to slow down or even crash. Memory leaks can be a significant problem in long-running applications and can lead to performance degradation.

Here are some common causes of memory leaks and strategies to avoid them:

1. **Failure to deallocate memory:** When a program allocates memory dynamically (e.g., using **malloc()** in C or C++), it should release the memory when it's no longer needed. Failing to do so results in memory leaks. To avoid this, always pair memory allocation functions with corresponding deallocation functions (e.g., **free()** in C or C++).
2. **Cyclic references in garbage-collected languages:** In languages with garbage collection (e.g., Java, Python), cyclic references can prevent objects from being properly reclaimed by the garbage collector. To avoid this, break cyclic references or use weak references when necessary.
3. **Unclosed resources:** Besides memory, other resources like file handles, database connections, or network sockets need to be explicitly closed when no longer needed. Failing to close these resources can lead to resource leaks. Use try-finally blocks or use resource management libraries to ensure proper cleanup.
4. **Improper cache management:** Caching data in memory can improve performance, but if not managed properly, it can lead to memory leaks. Implement cache eviction policies to remove items from the cache when they're no longer relevant.
5. **Use automated tools and profilers:** Many development environments and programming languages provide tools and profilers to detect memory leaks. These tools can help identify areas in your code where memory is allocated but not deallocated. Familiarize yourself with the memory profiling tools available for your language or framework.
6. **Static code analysis:** Consider using static code analysis tools that can identify potential memory leaks and other issues during the development and testing phases. These tools can provide early warnings and help you write more robust code.
7. **Unit testing and integration testing:** Incorporate memory leak detection into your testing process. Create test cases specifically designed to identify memory leaks. Automated tests can catch memory leaks early in the development cycle.
8. **Code reviews:** Conduct code reviews with your team to ensure that memory management practices are followed consistently and to catch potential issues before they become problematic.
9. **Regular monitoring:** In long-running applications, monitor memory usage over time. Implement memory profiling and monitoring in production to catch memory leaks in real-world scenarios. Tools like monitoring systems and log analysis can help in identifying memory-related issues.

Avoiding memory leaks is essential for creating stable and efficient software. By following best practices and using appropriate tools, you can minimize the risk of memory leaks in your applications.

**If the application is slow then how I can improve it?**

Improving the performance of a slow application can be a challenging task, but there are various strategies and techniques you can employ to enhance its speed and responsiveness. Here are some steps you can take to address performance issues in your application:

1. **Identify Performance Bottlenecks:**

**- Use profiling tools:** Profiling tools can help you identify performance bottlenecks in your code. They provide insights into which parts of your application are consuming the most resources and taking the most time.

**- Monitor resource usage:** Keep an eye on CPU, memory, disk, and network usage to determine if any particular resource is being exhausted.

**2. Optimize Code and Algorithms:**

**- Review your code:** Look for inefficient algorithms, unnecessary loops, and redundant code. Simplify and optimize your code where possible.

**- Use appropriate data structures:** Choose the right data structures and algorithms for your specific use case. For example, use efficient sorting algorithms or data structures like hash tables when needed.

**- Minimize database queries:** Reduce the number of database queries by using efficient SQL queries, caching, and reducing redundant database calls.

**3. Improve Database Performance:**

**- Indexing:** Properly index your database tables to speed up data retrieval operations. Ensure that indexes are being used effectively by your queries.

**- Database optimization:** Consider database tuning, denormalization, and caching strategies to reduce the load on your database.

**4. Frontend Optimization:**

**- Reduce client-side rendering:** Minimize the amount of work that needs to be done on the client-side by optimizing your HTML, CSS, and JavaScript.

**- Compress and optimize assets:** Compress images, JavaScript, and CSS files to reduce load times. Use content delivery networks (CDNs) for delivering static assets.

**5. Concurrency and Parallelism:**

**- Utilize multithreading or multiprocessing:** Use concurrent programming techniques to take advantage of multi-core processors and perform tasks in parallel.

**- Asynchronous programming:** Use asynchronous techniques to avoid blocking operations, especially in I/O-bound tasks.

**6. Caching:**

**- Implement caching:** Use in-memory caching for frequently accessed data to reduce the need for repeated expensive computations or database queries.

**7. Load Balancing:**

**- Distribute load:** If your application is under heavy load, consider using load balancing to distribute traffic across multiple servers or instances.

**8. Database Sharding:**

**- Consider database sharding:** If your application's database is a bottleneck, you can shard or partition the data across multiple database servers to improve scalability.

**9. Content Delivery Networks (CDNs):**

**- Use CDNs:** Offload static content to a CDN, which can deliver content from servers located closer to the end users, reducing latency.

**10. Optimize Network Usage:**

**- Minimize unnecessary data transfer:** Reduce the amount of data transferred over the network by using compression and minimizing unnecessary requests.

**- Use efficient protocols:** Select efficient network protocols for communication, such as HTTP/2 or HTTP/3.

**11. Security Considerations:**

- Ensure that security measures, such as input validation, are not causing performance degradation.

**12. Regular Testing:**

**- Perform load testing:** Test your application's performance under heavy load to identify bottlenecks and scalability issues.

**- Continuous monitoring:** Continuously monitor your application's performance and make adjustments as necessary.

**13. Scalability:**

**- Plan for scalability:** Design your application to be horizontally scalable, so you can add more resources or servers as needed.

Remember that performance optimization is an ongoing process. Regularly review and refine your application's performance as it evolves and as new usage patterns emerge. Additionally, prioritize optimizations based on the specific bottlenecks and issues affecting your application's performance.

**There are 100 elements in array in random array we have to find those elements whose left side element need to smaller and right side element needs to be greater using C#.**

You can find the elements in an array where the element on the left side is smaller and the element on the right side is greater in C# by iterating through the array and checking the elements. Here's a C# code snippet to achieve this:

To minimize the time and space complexity for finding elements in an array where the element on the left is smaller and the element on the right is greater, you can optimize the algorithm as follows:

**1. Optimized Algorithm:**

- Start from the left side of the array and maintain the maximum element found so far.

- Iterate through the array and for each element `arr[i]`:

- Check if it's greater than the maximum element found so far. If it is, set it as the new maximum.

- If `arr[i]` is not the maximum element, check if it's greater than or equal to the elements to its right.

- If both conditions are met, `arr[i]` satisfies the criteria and can be added to the result.

Here's the C# code for the optimized algorithm:

using System;

using System.Collections.Generic;

class Program

{

static List<int> FindElements(int[] arr)

{

List<int> result = new List<int>();

int n = arr.Length;

int maxSoFar = arr[0];

for (int i = 1; i < n - 1; i++)

{

if (arr[i] > maxSoFar)

{

maxSoFar = arr[i];

if (arr[i] >= arr[i + 1])

{

result.Add(arr[i]);

}

}

}

return result;

}

static void Main(string[] args)

{

int[] array = { 4, 3, 7, 6, 8, 9, 5 };

List<int> elements = FindElements(array);

Console.WriteLine("Elements where left is smaller and right is greater:");

foreach (int element in elements)

{

Console.Write(element + " ");

}

}

}

This optimized algorithm has a time complexity of O(n) and uses minimal additional space, resulting in improved performance compared to the previous approach.

**How to manage the threads in C#?**

Managing threads in C# involves controlling the execution of multiple threads within a program to achieve concurrent or parallel processing. There are various ways to manage threads in C#. Here are some key concepts and techniques:

**1. Thread Class:**

- You can create and manage threads using the `System.Threading.Thread` class. You can create a new thread by instantiating this class and passing a delegate method that contains the code to be executed by the thread.

Thread myThread = new Thread(MyThreadFunction);

myThread.Start();

**2. Thread Pool:**

- .NET provides a thread pool that manages a pool of worker threads, which is useful for managing and reusing threads efficiently. You can queue work items using the `ThreadPool.QueueUserWorkItem` method.

ThreadPool.QueueUserWorkItem(MyThreadFunction);

**3. Task Parallel Library (TPL):**

- TPL is a higher-level abstraction for managing threads and parallelism in C#. It simplifies the management of asynchronous and parallel operations by providing tasks. You can use the `Task` class to represent and manage asynchronous operations.

Task.Run(() => MyAsyncFunction());

**4. Async and Await:**

- The `async` and `await` keywords in C# simplify asynchronous programming, allowing you to write code that can yield control and return to the caller without blocking the main thread. This is especially useful for I/O-bound operations.

async Task MyAsyncFunction()

{

// Asynchronous code here

}

**5. Thread Synchronization:**

- When dealing with shared resources, you need to manage thread synchronization to avoid race conditions. C# provides tools like `lock`, `Monitor`, and `Mutex` for synchronizing access to shared data.

lock (lockObject)

{

// Critical section

}

**6. Thread Safety and Concurrency Patterns:**

- You should design your code with thread safety in mind. Implement common concurrency patterns like reader-writer locks, producer-consumer queues, and parallel loops for efficient thread management.

**7. Cancellation:**

- You can cancel long-running threads or tasks using cancellation tokens, which are useful for gracefully terminating threads or tasks.

CancellationTokenSource cts = new CancellationTokenSource();

CancellationToken token = cts.Token;

**8. Thread Priorities:**

- You can assign priorities to threads, although these are only hints to the operating system. Use `Thread.Priority` to set thread priorities.

myThread.Priority = ThreadPriority.AboveNormal;

**9. Thread Lifecycle:**

- Be aware of the various states in the lifecycle of a thread, such as the running state, sleeping state, and terminated state. Understand how to start, pause, and stop threads appropriately.

**10. Exception Handling:**

- Properly handle exceptions in your threaded code to avoid unhandled exceptions that can crash the application.

Managing threads in C# can be complex, and you should carefully consider the requirements of your application to choose the appropriate threading model and synchronization mechanisms. Be cautious when dealing with shared resources to avoid race conditions and deadlocks. Additionally, consider using modern asynchronous programming techniques to improve the responsiveness of your application.

**What is Thread pool in C#?**

In C#, a thread pool is a managed pool of worker threads that are available for executing tasks or work items asynchronously. It is a mechanism provided by the .NET Framework to efficiently manage and reuse threads, allowing you to perform parallel or concurrent operations without the overhead of creating and destroying threads for each task. Thread pools are commonly used for improving the performance and responsiveness of applications that require asynchronous or parallel processing.

Key features and characteristics of the C# thread pool include:

**1. Automatic Thread Management:** The thread pool automatically manages the creation, reuse, and recycling of threads, reducing the overhead associated with thread creation and destruction.

**2. Limited Thread Count:** The thread pool maintains a fixed number of worker threads, which typically depends on the system's configuration, such as the number of CPU cores. This ensures that the application does not create an excessive number of threads that could lead to resource exhaustion.

**3. Queueing Mechanism:** Tasks or work items are placed in a queue, and available worker threads from the pool pick up tasks to execute. This queuing mechanism helps ensure that tasks are executed in a controlled and efficient manner.

**4. Task Execution:** You can queue work items for execution in the thread pool using the `ThreadPool.QueueUserWorkItem` method. This method takes a delegate (typically a `WaitCallback`) representing the work item to be executed.

ThreadPool.QueueUserWorkItem(MyWorkerMethod);

**5. Cancellation and Asynchronous Execution:** The thread pool allows you to cancel tasks using cancellation tokens and supports asynchronous programming patterns. You can use `Task.Run` to queue asynchronous tasks.

Task.Run(async () => await MyAsyncMethod());

**6. Completion Callbacks:** You can specify a callback method that gets executed upon the completion of a work item, allowing you to perform cleanup or other actions after a task finishes.

ThreadPool.QueueUserWorkItem(MyWorkerMethod, null, MyCompletionCallback);

**7. Thread Prioritization:** Threads in the thread pool can have priorities assigned to them, although these are just hints to the operating system.

ThreadPool.SetMinThreads(4, 4);

ThreadPool.SetMaxThreads(10, 10);

**8. Load Balancing:** The thread pool automatically balances the workload among the available threads, which helps prevent overloading a single thread.

The thread pool is a convenient way to parallelize work in C# applications and is commonly used for tasks such as I/O-bound operations, background processing, and handling multiple concurrent requests in web applications. However, it is important to use thread pool threads judiciously and to be aware of potential issues, such as blocking operations, when working with the thread pool.

**What is delegate and provide a code sniffed for C#?**

A delegate in C# is a type that represents a reference to a method with a specific signature. Delegates are often used to define and encapsulate method signatures, allowing methods to be treated as first-class objects. They enable the implementation of callback mechanisms, event handling, and dynamic method invocation.

Here's a simple code snippet in C# that demonstrates the use of a delegate:

using System;

delegate int MathOperation(int a, int b);

class Program

{

// Sample methods that match the delegate's signature

static int Add(int a, int b)

{

return a + b;

}

static int Subtract(int a, int b)

{

return a - b;

}

static void Main(string[] args)

{

// Create delegate instances and associate them with methods

MathOperation addDelegate = new MathOperation(Add);

MathOperation subtractDelegate = new MathOperation(Subtract);

// Use the delegate instances to call the associated methods

int result1 = addDelegate(5, 3); // Calls the Add method

int result2 = subtractDelegate(10, 4); // Calls the Subtract method

Console.WriteLine($"Addition Result: {result1}");

Console.WriteLine($"Subtraction Result: {result2}");

}

}

In this code snippet:

1. We define a delegate named `MathOperation` that represents a method taking two integers as parameters and returning an integer. This delegate acts as a blueprint for methods with this signature.

2. We define two static methods, `Add` and `Subtract`, which match the delegate's signature.

3. Inside the `Main` method, we create instances of the `MathOperation` delegate and associate them with the `Add` and `Subtract` methods, respectively.

4. We then use the delegate instances to call the associated methods dynamically. The delegate instances `addDelegate` and `subtractDelegate` act as references to the methods they're associated with.

The code demonstrates how delegates allow you to switch between different methods at runtime by using a common delegate type, making it easy to create flexible and extensible code, particularly when you need callback mechanisms, event handling, or dynamic method invocation.

**What is Dependency Injection?**

Dependency Injection (DI) is a design pattern and a technique used in software development to achieve the separation of concerns, improve code maintainability, and promote the reusability of components within an application. Dependency Injection is particularly prevalent in object-oriented programming, including languages like C#, Java, and others. It's commonly used in frameworks and libraries, especially in the context of Inversion of Control (IoC) containers.

The fundamental concept of Dependency Injection is to decouple components and their dependencies. In this pattern, instead of a component creating its dependencies, the dependencies are provided to the component from the outside. This allows for greater flexibility and testability of the code. There are three primary forms of Dependency Injection:

**1. Constructor Injection:**

- In this form of DI, a class's dependencies are passed into the class via its constructor. The class doesn't create its own dependencies but relies on external code to provide them.

public class OrderService

{

private readonly IOrderRepository \_repository;

public OrderService(IOrderRepository repository)

{

\_repository = repository;

}

}

**2. Property Injection:**

- In property injection, a class's dependencies are set using public properties. This allows for optional dependencies that are set after an object is constructed.

public class OrderService

{

public IOrderRepository Repository { get; set; }

}

**3. Method Injection:**

- In method injection, a class's dependencies are passed as arguments to its methods, rather than being set in the constructor. This is useful when specific methods need certain dependencies.

public class OrderService

{

public void ProcessOrder(Order order, IOrderProcessor processor)

{

// Method-specific injection

}

}

The primary benefits of Dependency Injection include:

- Decoupling: It decouples components, making them less dependent on specific implementations. This promotes flexibility and easier maintenance of the code.

- Testability: It simplifies unit testing because dependencies can be easily replaced with mock objects or stubs, allowing for isolated testing of components.

- Reusability: It enables the reuse of components across different parts of the application or even in different projects, as long as the correct dependencies are provided.

- Configurability: It allows for easy configuration and swapping of components at runtime, facilitating the use of different implementations or configurations.

To implement Dependency Injection in C# or other object-oriented languages, you can manually pass dependencies through constructors or use frameworks and libraries that provide IoC containers, such as Microsoft's Dependency Injection container in ASP.NET Core or third-party libraries like Autofac, Ninject, or Unity. These containers manage the creation and injection of dependencies, making it more convenient to use the DI pattern.

**What is Multi-Threading in C#? Implement Multi-Threading?**

Multi-threading in C# is a programming technique that allows an application to have multiple threads of execution running concurrently. Each thread represents a separate path of execution through the code and can perform tasks independently of the main thread. This can lead to better performance and responsiveness in applications, especially when handling tasks that are computationally or I/O intensive. C# provides robust support for multi-threading through the System.Threading namespace.

Here's a basic example of how to implement multi-threading in C#:

using System;

using System.Threading;

class Program

{

static void Main()

{

// Create and start two separate threads

Thread thread1 = new Thread(DoWork1);

Thread thread2 = new Thread(DoWork2);

thread1.Start();

thread2.Start();

// Wait for both threads to finish

thread1.Join();

thread2.Join();

Console.WriteLine("Both threads have completed.");

}

static void DoWork1()

{

for (int i = 0; i < 5; i++)

{

Console.WriteLine("Thread 1: Step " + i);

Thread.Sleep(1000); // Simulate some work

}

}

static void DoWork2()

{

for (int i = 0; i < 5; i++)

{

Console.WriteLine("Thread 2: Step " + i);

Thread.Sleep(800); // Simulate some work

}

}

}

In this example:

1. We import the `System.Threading` namespace.

2. We define two methods, `DoWork1` and `DoWork2`, which simulate some work by printing messages and using `Thread.Sleep` to pause the threads.

3. In the `Main` method, we create two separate threads (`thread1` and `thread2`) and start them using the `Start` method.

4. We use the `Join` method to wait for both threads to finish their work.

5. After both threads have completed, we print a message to the console.

Multi-threading can be more complex in real-world applications, especially when dealing with shared resources that need to be synchronized between threads. You should be cautious when working with multi-threading to prevent race conditions, deadlocks, and other synchronization issues. C# provides various synchronization primitives like `Monitor`, `Mutex`, `Semaphore`, and more to help manage thread synchronization. Additionally, in modern C#, you can use `async` and `await` to simplify asynchronous programming and avoid some of the complexities of low-level multi-threading.

**What is Task Parallel Library in C#? Implement Task Parallel Library.**

The Task Parallel Library (TPL) in C# is a set of .NET libraries and APIs that provide a higher-level abstraction for parallel and concurrent programming. TPL makes it easier to write multi-threaded and parallel code, allowing developers to take advantage of modern multi-core processors without dealing with the low-level complexities of managing threads. TPL is part of the System.Threading.Tasks namespace and provides a more developer-friendly approach to parallelism in C#.

Here's a simple example of how to implement the Task Parallel Library in C#:

using System;

using System.Threading.Tasks;

class Program

{

static void Main()

{

// Create and start two parallel tasks

Task task1 = Task.Run(() => DoWork1());

Task task2 = Task.Run(() => DoWork2());

// Wait for both tasks to complete

Task.WhenAll(task1, task2).Wait();

Console.WriteLine("Both tasks have completed.");

}

static void DoWork1()

{

for (int i = 0; i < 5; i++)

{

Console.WriteLine("Task 1: Step " + i);

Task.Delay(1000).Wait(); // Simulate some work

}

}

static void DoWork2()

{

for (int i = 0; i < 5; i++)

{

Console.WriteLine("Task 2: Step " + i);

Task.Delay(800).Wait(); // Simulate some work

}

}

}

In this example:

1. We use the `System.Threading.Tasks` namespace.

2. We create two tasks, `task1` and `task2`, using `Task.Run()`. These tasks execute the `DoWork1` and `DoWork2` methods, respectively, in parallel.

3. We use `Task.WhenAll` to wait for both tasks to complete.

4. Inside the `DoWork1` and `DoWork2` methods, we simulate some work using `Task.Delay()` to pause the tasks asynchronously.

5. After both tasks have completed, we print a message to the console.

The TPL provides a higher level of abstraction for managing parallelism and concurrency, making it easier to write efficient and safe multi-threaded code. It also offers features like cancellation, exception handling, and the ability to work with asynchronous operations using `async` and `await`.

In real-world applications, you can use TPL to parallelize tasks such as data processing, I/O operations, and more to take advantage of the full processing power of modern multi-core CPUs.

**What is Design Pattern? Types of Design Pattern.**

A design pattern is a general reusable solution to a common problem that occurs in software design. Design patterns are not complete designs or templates, but rather guidelines or best practices that help developers solve specific design problems. They represent the knowledge and experience of skilled software designers and provide a blueprint for building maintainable, efficient, and scalable software systems. Using design patterns can lead to more structured and organized code, better code reusability, and easier maintenance.

There are several categories of design patterns, but some of the most commonly recognized types include:

**1. Creational Patterns:**

**- Singleton Pattern:** Ensures that a class has only one instance and provides a global point of access to that instance.

**- Factory Method Pattern:** Defines an interface for creating an object but lets subclasses alter the type of objects that will be created.

**- Abstract Factory Pattern:** Provides an interface for creating families of related or dependent objects without specifying their concrete classes.

**- Builder Pattern:** Separates the construction of a complex object from its representation, allowing the same construction process to create different representations.

**2. Structural Patterns:**

**- Adapter Pattern:** Allows the interface of an existing class to be used as another interface.

**- Decorator Pattern:** Attaches additional responsibilities to an object dynamically, providing a flexible alternative to subclassing.

**- Composite Pattern:** Composes objects into tree structures to represent part-whole hierarchies. It lets clients treat individual objects and compositions of objects uniformly.

**- Proxy Pattern:** Provides a surrogate or placeholder for another object to control access to it.

**3. Behavioral Patterns:**

**- Observer Pattern:** Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

**- Strategy Pattern:** Defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

**- Command Pattern:** Encapsulates a request as an object, thereby allowing for parameterization of clients with queues, requests, and operations.

**- State Pattern:** Allows an object to alter its behavior when its internal state changes. The object will appear to change its class.

**4. Architectural Patterns:**

**- Model-View-Controller (MVC):** Separates an application into three interconnected components: the model (business logic), the view (presentation and user interface), and the controller (user input and system management).

**- Model-View-ViewModel (MVVM):** Similar to MVC but tailored for modern user interfaces and data-driven applications.

**- Layered Architecture:** Divides an application into multiple layers (e.g., presentation, business, and data layers) to separate concerns and maintain modularity.

**5. Concurrency Patterns:**

**- Producer-Consumer Pattern:** Deals with the efficient synchronization of producers and consumers of data or resources.

**- Mutex Pattern:** Helps manage mutual exclusion to ensure that only one thread can access a resource at a time.

**- Read-Write Lock Pattern:** Provides concurrent read access to a resource but exclusive write access.

**6. Idiomatic Patterns:**

- These patterns are often based on language-specific features and coding conventions. For example, the Dispose pattern in C# for managing resources or the RAII (Resource Acquisition Is Initialization) idiom in C++.

Design patterns are not strict templates to be followed blindly; they should be applied thoughtfully based on the specific problem you're trying to solve. The goal is to promote best practices, increase code maintainability, and enhance the flexibility and scalability of your software.

### What is Thread Pooling in C#?

In C#, a Thread Pool is a group of threads. These threads are used to do work without interfering with the principal thread's operation.

**What is C# Aggregation?**

In C#, aggregation is a process in which one class defines another class as any entity reference. It is another way to reuse the class. It is a form of association that represents HAS-A relationship.S

**Polymorphism:** Polymorphism is the concept where an object behaves differently in different situations. There are two types of polymorphism –

**Compile time polymorphism** is achieved by method overloading.

**Runtime polymorphism** is implemented when we have “IS-A” relationship between objects and is achieved by method overriding.

### Ques: What is the difference between “throw” and “throw ex” in C#?

The difference between “throw” and “throw ex” is that “throw” preserves the stack trace (the stack trace will point to the method that caused the exception in the first place) while ”throw ex” does not preserve the stack trace (we will lose the information about the method that caused the exception in the first place. It will seem like the exception was thrown from the place of its catching and re-throwing).

# **Ques: Difference between UNION and UNION ALL**

## **UNION vs. UNION ALL**

|  |  |  |
| --- | --- | --- |
| **Parameter** | **UNION** | **UNION ALL** |
| **Definition** | Combines the result from the multiple table and return the distinct records into a single result set. | Combines the result from the multiple table and return all the record into a single result set. |
| **Removal of dulicates** | It has a default feature to eliminate duplicate rows from the table. | It can’t eliminate the duplicate rows from the table. |
| **Performance** | Slow performance as it removes duplicate rows from the output. | Fast performance as compare to UNION. |
| **Syntax** | SELECT column\_names FROM table\_1 WHERE conditions UNION SELECT column\_names FROM table\_2 WHERE conditions | SELECT column\_names FROM table\_1 WHERE conditions UNION ALL SELECT column\_names FROM table\_2 WHERE conditions |

**Ques: What is conceptual and storage model in Entity Framework?**

# **Entity Framework Architecture**

Let's look at the components of the architecture individually.

**EDM (Entity Data Model):**EDM consists of three main parts - Conceptual model, Mapping and Storage model.

**Conceptual Model:**The conceptual model contains the model classes and their relationships. This will be independent from your database table design.

**Storage Model:** The storage model is the database design model which includes tables, views, stored procedures, and their relationships and keys.

**Mapping:**Mapping consists of information about how the conceptual model is mapped to the storage model.

# **What is an Entity in Entity Framework?**

<https://www.entityframeworktutorial.net/entityframework6/entityframework-architecture.aspx>

An entity in Entity Framework is a class that maps to a database table. This class must be included as a DbSet<TEntity> type property in the DbContext class. EF API maps each entity to a table and each property of an entity to a column in the database.

For example, the following Student, and Grade are domain classes in the school application.

public class Student

{

public int StudentID { get; set; }

public string StudentName { get; set; }

public DateTime? DateOfBirth { get; set; }

public byte[] Photo { get; set; }

public decimal Height { get; set; }

public float Weight { get; set; }

public Grade Grade { get; set; }

}

public class Grade

{

public int GradeId { get; set; }

public string GradeName { get; set; }

public string Section { get; set; }

public ICollection<Student> Students { get; set; }

}

The above classes become entities when they are included as DbSet<TEntity> properties in a context class (the class which derives from DbContext), as shown below.

public class SchoolContext : DbContext

{

public SchoolContext()

{

}

public DbSet<Student> Students { get; set; }

public DbSet<Grade> Grades { get; set; }

}

In the above context class, Students, and Grades properties of type DbSet<TEntity> are called entity sets. The Student, and Grade are entities. EF API will create the Students and Grades tables in the database, as shown below.

**Ques: What is use of DbContext and DbSet in Entity Framework?**

In Entity Framework, `DbContext` and `DbSet` are important components that play key roles in working with a database.

**1. DbContext:**

- `DbContext` is a crucial class in Entity Framework that represents a session with the database, allowing you to query and interact with the database.

- It is responsible for tracking changes to entities, managing the connection to the database, and coordinating the overall interaction between your application and the database.

Example of a simple `DbContext` class:

public class AppDbContext : DbContext

{

public DbSet<User> Users { get; set; }

public DbSet<Order> Orders { get; set; }

// Other configuration and methods can be added here

}

In this example, `AppDbContext` is a derived class from `DbContext`, and it exposes `DbSet` properties (`Users` and `Orders`) for each entity type you want to interact with in the database.

**2. DbSet:**

- `DbSet` is a collection of entities of a specific type in the context of Entity Framework. It represents the entities of a particular type in the database and provides methods for querying, adding, updating, and deleting entities.

- Each `DbSet` property in a `DbContext` corresponds to a table in the database.

Example of using `DbSet` in a DbContext:

public class AppDbContext : DbContext

{

public DbSet<Product> Products { get; set; }

public DbSet<Category> Categories { get; set; }

// Other configuration and methods can be added here

}

In this example, `Products` and `Categories` are `DbSet` properties, indicating that the `AppDbContext` will interact with tables named "Products" and "Categories" in the database.

When you create a `DbContext` instance and use the `DbSet` properties, Entity Framework takes care of generating SQL queries and managing the connection to the database, making it easier for developers to interact with the database using object-oriented principles.

**What is Lazy Loading, and how does it work in Entity Framework?**

**Answer:** Lazy Loading is a feature in Entity Framework that loads related entities (navigation properties) from the database on-demand, as you access them. It helps to improve performance by not loading related entities until they are actually needed. Lazy Loading is enabled by default in Entity Framework.

**Explain Eager Loading in Entity Framework.**

**Answer:** Eager Loading is the opposite of Lazy Loading. It loads related entities along with the main entity in a single query to the database. This can help reduce the number of database round-trips and improve performance. Eager Loading is accomplished using the **Include** method in Entity Framework.