# Short Answer:

Answer the following questions with complete sentences in your own words. You are encouraged to conduct your own research online or through other methods before answering the questions. If you research online, please consult multiple sources before you write down your answers. You are expected to be able to explain your answers in detail (Provide examples for each question).

1. What is LINQ and What’s the advantage of using LINQ?

What is LINQ?

● LINQ(Language Integrated Query) provides us with common query syntax

which allows us to query the data from various data sources.

● It was introduced by Microsoft with .NET Framework 3.5 and C# 3.0 and is

available in System.Linq namespace.

● It supports a consistent query experience(LINQ Provider)

○ LINQ to objects

○ LINQ to SQL

○ LINQ to XML

○ …

LINQ (Language Integrated Query) is a set of language features in C# and Visual Basic that enables developers to write queries against data sources in a variety of formats, such as in-memory collections, SQL databases, and XML documents. LINQ provides a consistent syntax for querying and manipulating data, regardless of the data source.

The main advantage of using LINQ is that it enables developers to write expressive, declarative code to query and manipulate data. This can make the code easier to read and maintain, as it more closely resembles the structure of the data being queried. LINQ also provides a range of built-in operators and functions that allow developers to easily filter, sort, and transform data in their queries. Additionally, LINQ is integrated with the .NET framework and can be used in a variety of applications, including desktop, web, and mobile.

1. What are the different ways to write LINQ query syntax?

● Query Syntax ● Method Syntax ● Mixed Syntax (Query + Method)



1. . Explain what are LINQ providers and what is the purpose of LINQ providers in LINQ?

LINQ Providers

❑ LINQ to objects (No provider needed but a type of LINQ): allows us to query in-memory objects from an

array, collection and generics types.

● LINQ to XML(XLINQ): works with XML documents.

● LINQ to SQL(DLINQ): works with the SQL Server database.

● LINQ to Datasets: provides us the flexibility to query data cached in a Dataset.

● LINQ to Entities: is used to query any database(including SQL Server, Oracle, MySQL

etc.

Query Syntax vs. Method Syntax

Most queries in the introductory Language Integrated Query (LINQ) documentation are written by using the LINQ

declarative query syntax. However, the query syntax must be translated into method calls for the .NET common language

runtime (CLR) when the code is compiled. These method calls invoke the standard query operators, which have names

such as Where, Select, GroupBy, Join, Max, and Average. You can call them directly by using method syntax instead of

query syntax.

There will be some queries that must be expressed as method calls. you must use a method call to express a query that

retrieves the number of elements that match a specified condition. You also must use a method call for a query that

retrieves the element that has the maximum value in a source sequence.

Text

Description automatically generated

LINQ (Language-Integrated Query) is a .NET framework feature that enables developers to write queries against data sources in a uniform and expressive syntax. LINQ provides a set of extension methods that allow developers to query data sources such as arrays, lists, databases, and XML documents using a consistent syntax.

LINQ providers are components that enable LINQ to work with a specific type of data source. LINQ providers translate LINQ queries into the native syntax of the data source, allowing developers to write queries using LINQ syntax without worrying about the underlying data source.

For example, the LINQ to Objects provider allows developers to write LINQ queries against in-memory collections such as lists and arrays. The LINQ to SQL provider allows developers to write LINQ queries against a SQL Server database. The LINQ to XML provider allows developers to write LINQ queries against XML documents.

The purpose of LINQ providers is to enable developers to use LINQ syntax to query a wide variety of data sources in a consistent and expressive way. By abstracting the underlying data source, LINQ providers make it easier for developers to write and maintain code that works with different data sources.

I hope this helps! Let me know if you have any other questions.

1. What are the basic steps of executing LINQ Query?

● All LINQ query operations consist of three distinct actions:

1. Obtain the data source

2. Create the query

The query expression contains three clauses: from, where and select

The from clause specifies the data source, where clause applies the filter, and the select clause

specifies the type of the returned elements.

Text

Description automatically generated with medium confidence

3. Execute the query

The actual execution of the query is deferred until you iterate over the query variable in a foreach statement.

Text

Description automatically generated

Whole example:

Graphical user interface, text, application

Description automatically generated

1. What is the diamond problem? And how can we resolve the problem?

The diamond problem, also known as the "deadly diamond of death," is a scenario that can arise in object-oriented programming when multiple inheritance is allowed and two classes have a common base class. **Consider the following example in C#:**

class A { }

class B : A { }

class C : A { }

class D : B, C { }

In this example, class D is derived from both B and C, which are both derived from A. This creates a diamond-shaped inheritance hierarchy, with A at the top and D at the bottom. The problem occurs when D tries to access a member of A that is overridden by both B and C. In this case, it is not clear which version of the member D should access.

One way to resolve the diamond problem is to use virtual inheritance, also known as the "diamond problem workaround." With virtual inheritance, the common base class is specified as a virtual base class, and the derived classes B and C are specified as inheriting virtually from A. This ensures that there is only one instance of the base class A in the inheritance hierarchy, and D can access it unambiguously.

Another way to resolve the diamond problem is to use interface-based inheritance, where the common base class A is defined as an interface rather than a class. Derived classes B and C can then implement the interface, and class D can implement both B and C. This avoids the need for multiple inheritance and the potential conflicts that it can cause.

1. What is an interface and what is an abstract class? What are the differences between them?

In C#, an interface is used to define the outer abilities of a class. An abstract class is used to define a class’s actual identity, and it is used as the object or the same type.

In C#, an interface is used if various implementations only shared method signatures.

Graphical user interface, text, table

Description automatically generated

In object-oriented programming, an interface is a set of related methods that a class can implement. An interface defines the signature of the methods, including the name, return type, and parameters, but does not provide an implementation for them. A class that implements an interface is required to implement all of the methods defined in the interface.

An abstract class, on the other hand, is a class that cannot be instantiated on its own and is meant to serve as a base class for one or more derived classes. An abstract class can contain both abstract methods, which are methods that have a signature but no implementation, and concrete methods, which have a full implementation.

There are a few key differences between interfaces and abstract classes:

Interfaces only contain method signatures and no implementation, while abstract classes can contain both abstract and concrete methods.

A class can implement multiple interfaces, but can only inherit from a single abstract class.

Interfaces are used to define a set of related behaviors that a class can implement, while abstract classes are used to provide a common base implementation for derived classes.

Interfaces are more flexible than abstract classes because they allow a class to implement multiple unrelated behaviors, while abstract classes require a class to inherit a single implementation.

It's important to choose the appropriate tool for the job. If you want to define a set of behaviors that a class can implement, an interface is the appropriate choice. If you want to provide a common base implementation for derived classes, an abstract class is a better choice.

1. Why SELECT clause comes after FROM clause in LINQ?

The from clause specifies the data source, the where clause applies the filter, and the select clause

specifies the type of the returned elements.

The order of the clauses in a LINQ query follows the structure of a SQL query, with the SELECT clause coming after the FROM clause. The FROM clause specifies the data source for the query, and the SELECT clause specifies the projection(determine the return type), or the shape of the data that is returned.

var query = from s in students

select s.Name;

In this example, the **students** collection is the data source, and the **select s.Name** clause specifies that the name of each student should be included in the results. The structure of the query follows the structure of a SQL SELECT statement, with the FROM clause specifying the data source and the SELECT clause specifying the projection.

The order of the clauses in a LINQ query is important because it determines the logical flow of the query. The FROM clause is evaluated first, and it specifies the source of the data that is being queried. The SELECT clause is then applied to the results of the FROM clause, and it specifies the shape of the data that is returned. This allows developers to write expressive, declarative code to query and manipulate data in a consistent and intuitive way.

1. What is the anonymous type?

An anonymous type is a type that is generated automatically by the compiler and does not have a name. It is used to create an object with a set of properties, without the need to explicitly define a class for the object.

To create an anonymous type in C#, you can use the new operator and specify the properties of the object using a comma-separated list of property names and values. The type of the properties is inferred by the compiler based on the values that are assigned to them. Here is an example of creating an anonymous type in C#:

var person = new { Name = "John", Age = 30 };

In this example, the anonymous type is created with two properties: Name, which is of type string, and Age, which is of type int. The type of the anonymous object is inferred by the compiler based on the values that are assigned to the properties.

Anonymous types are often used in LINQ queries to create temporary objects that are used to hold the results of the query. They are also useful for creating simple objects without the need to define a full class. However, anonymous types have some limitations, such as not being able to define methods or implement interfaces.

1. What is Encapsulation? How does C# implement it? And why do we need encapsulation?

Encapsulation is a principle of object-oriented programming that refers to the bundling of data and behavior within a class. It is a way of hiding the implementation details of a class from the outside world, and providing a well-defined interface for interacting with the class.

In C#, encapsulation is implemented using access modifiers, such as **public**, **private**, and **protected**, which determine the visibility and accessibility of class members (fields, properties, methods, etc.). For example, a field or property that is marked as **private** is only accessible within the class, while a member that is marked as **public** is accessible from outside the class.

Encapsulation is an important concept in software development because it helps to reduce complexity and improve the maintainability of code. By encapsulating the implementation details of a class within the class itself, we can hide the complexity of the class from the rest of the system and provide a simpler, more predictable interface for interacting with the class. This makes it easier to understand and modify the class, and can help to prevent unintended side effects when making changes to the class.

Encapsulation also promotes code reuse and modularity, as it allows developers to create self-contained, independent classes that can be used in different contexts and composed together to build larger systems. By encapsulating data and behavior within a class, we can create reusable, modular units of code that are easy to test and maintain.

1. What is Standard Query Operators?

● The standard query operators are the methods that form the LINQ

pattern. Most of these methods operate on sequences, where a

sequence is an object whose type implements the IEnumerable<T>

interface. The standard query operators provide query capabilities

including filtering, projection, aggregation, sorting and more.

● Query Expression Syntax for Standard Query Operators:

○ Cast, GroupBy, GroupJoin, Join, OrderBy, OrderByDescending, Select, SelectMany,

ThenBy, ThenByDescending, Where, and more

Graphical user interface, text, application, email

Description automatically generated

1. What are immediate execution and deferred execution? Giving some examples for each

Manners Of Execution

LINQ operators are divided into 2 categories:

● Deferred Execution

Deferred execution means that the operation is not performed at the point in the code where the query is declared. The operation is performed only when the query variable is enumerated, for example by using a foreach statement.

Almost all the standard query operators whose return type is IEnumerable<T>

Ex: Select(), SelectMany(), Where(), etc.

● Immediate Execution

Immediate execution means that the data source is read and the operation is performed at the point in the code where the query is declared.

Ex: ToArray(), ToList(), Aggregate Methods: etc.

Graphical user interface, text, application

Description automatically generated

1. When to use First() and when to use FirstOrDefault() with LINQ?

## First: return the first record when there is one or more records found; if not matched --> throw and exception

## FirstOrDefault: return the first record when there is one or more records found; if not matched --> assign the default value

**First()** and **FirstOrDefault()** are extension methods in LINQ that are used to retrieve the first element of a sequence. Both methods take a predicate as an optional parameter, which is a function that specifies the condition that the element must satisfy in order to be returned.

The main difference between **First()** and **FirstOrDefault()** is the way they behave when no element in the sequence satisfies the predicate or when the sequence is empty. **First()** will throw an exception if no element in the sequence satisfies the predicate or if the sequence is empty, while **FirstOrDefault()** will return the default value for the type of the elements in the sequence.

Here is an example of using **First()** and **FirstOrDefault()** in a LINQ query:

var students = new List<Student>

{

new Student { Name = "John", Age = 20 },

new Student { Name = "Jane", Age = 22 },

new Student { Name = "Jack", Age = 24 }

};

// Find the first student whose age is greater than 22

var student = students.First(s => s.Age > 22);

// Find the first student whose age is less than 22, or null if none exist

var student = students.FirstOrDefault(s => s.Age < 22);

In general, you should use **First()** if you expect the element to exist in the sequence and want to throw an exception if it does not. You should use **FirstOrDefault()** if you want to handle the case where the element does not exist or the sequence is empty.

1. Can we use ***this*** keyword in the constructor to invoke other constructors? Why?

A constructor can invoke another constructor in the same object by using the this keyword. This is known as constructor chaining.

There are a few reasons why you might want to use constructor chaining:

* To reuse common initialization code: If you have multiple constructors in a class and they all perform similar initialization tasks, you can avoid duplicating the code by using constructor chaining to call a single constructor that performs the common initialization.
* To enforce required arguments: You can use constructor chaining to ensure that certain arguments are always provided when an object is constructed. For example, you might have a class that requires a name and an age, and you can use constructor chaining to ensure that these arguments are always passed to the constructor.
* To provide default values: You can use constructor chaining to provide default values for arguments that are not required. For example, you might have a class that allows an optional email address, and you can use constructor chaining to provide a default email address if none is provided.

To use constructor chaining in C#, you can use the **this** keyword to invoke another constructor within the same class, and pass any required arguments as arguments to the **this** call. Here is an example of constructor chaining in C#:

public class Person

{

public string Name { get; set; }

public int Age { get; set; }

public string Email { get; set; }

public Person(string name, int age)

{

Name = name;

Age = age;

}

public Person(string name, int age, string email) : this(name, age)

{

Email = email;

}

}

In this example, the **Person** class has two constructors: one that takes a name and an age, and another that takes a name, an age, and an email address. The second constructor uses the **this** keyword to invoke the first constructor and pass the required **name** and **age** arguments, and then sets the **Email** property to the provided value. This allows the second constructor to reuse the common initialization code from the first constructor, while also providing the additional **email** argument.

1. What is a thread and what is a process?

Process

● A process has a self-contained execution environment.

A process generally has a complete, private set of basic runtime resources; in

particular, each process has its own memory space.

● Processes are often seen as synonymous with programs or applications

However, what the user sees as a single application may in fact be a set of

cooperating processes.

○ Eg.To facilitate communication between processes, most operating systems support Inter

Process Communication (IPC) resources, such as pipes and sockets

Thread

● A thread is a lightweight sub-process, the smallest unit of processing. It has a

separate path of execution.

● Threads are independent, if an exception occurs in one thread, it doesn't

affect other threads.

● In other words, exceptions thrown in one thread cannot be handled by

another thread.

Graphical user interface

Description automatically generated

A thread is a lightweight execution unit within a process that can run concurrently with other threads. A process is an instance of a running program, with its own memory space and resources.

In a computer with a single-core processor, only one thread can execute at a time, and the operating system switches between threads to give the appearance of concurrent execution. In a computer with a multi-core processor, multiple threads can execute concurrently on different cores.

Threads are useful for improving the performance and responsiveness of a program by allowing it to perform multiple tasks concurrently. For example, a program that performs a long-running computation in a single thread might appear unresponsive to the user until the computation is complete. By using multiple threads, the program can perform the computation in the background and continue to respond to user input in the foreground.

Processes, on the other hand, are used to isolate different programs and protect the system from errors or malicious behavior in one program. Each process has its own memory space and resources, which helps to prevent one process from accessing or modifying the memory or resources of another process.

In general, threads are used to improve the performance and responsiveness of a program within a single process, while processes are used to isolate different programs and protect the system.

Multi-Threading

● To enhance parallel processing

● To reduce response time to the user

○ Many servers use multithreads to achieve high performance

● To utilize the idle time of the CPU

○ Unit testing uses threads to run test cases in parallel

● Prioritize your work depending on priority

○ Computer games is a good example of multi-threading process(loading maps when you are

working on other things)

Thread Life Cycle

The life cycle of a thread starts when an object of the Thread class is created and ends when the thread is

terminated or completes execution.

• The Unstarted State − It is the situation when the instance of the thread is created but the Start method is

not called.

• The Runnable State − It is the situation when the thread is ready to run(the Start method has been called).

• The Running State – The thread is running and not stopped yet.

• The Not Runnable State − A thread is not executable, when

• Sleep method has been called

• Join method has been called

• Blocked by I/O operations

• The Dead State − It is the situation when the thread completes execution or is aborted by calling Abort()

Method

# 13. Talk about the Thread Life cycle.

Diagram

Description automatically generated

The life cycle of a thread starts when an object of the Thread class is created and ends when the thread is terminated or completes execution.

• The Unstarted State − It is the situation when the instance of the thread is created but the Start method is not called.

• The Runnable State − It is the situation when the thread is ready to run(the Start method has been called).

• The Running State – The thread is running and not stopped yet.

• The Not Runnable State − A thread is not executable, when

• Sleep method has been called

• Join method has been called

• Blocked by I/O operations

• The Dead State − It is the situation when the thread completes execution or is aborted by calling Abort()

method.

Not Runnable State

● Sleep() – Thread.Sleep() causes the current thread to suspend execution for a

specific milliseconds. Sleep() is a static method in Thread class.

● Join() – The Join() method blocks the calling thread until the thread represented by

this instance terminates.

○ Example: We create the Thread t in the main thread;

○ Calling t.Join() will cause the main thread to wait for thread t.

● Suspend() – Suspend() method is called to suspend the thread.（deprecated）

● Resume() – Resume() method is called to resume the suspended thread.(deprecated)

Diagram

Description automatically generated

Life Cycle of a thread

Unstarted state: When an instance of a Thread class is created, it is in the unstarted state, means the thread has not yet started to run when the thread is in this state. Or in other words Start() method is not called.

Thread thr = new Thread();

Here, thr is at unstarted state.

Runnable State: A thread that is ready to run is moved to runnable state. In this state, a thread might actually be running or it might be ready to run at any instant of time. It is the responsibility of the thread scheduler to give the thread, time to run. Or in other words, the Start() method is called.

Running State: A thread that is running. Or in other words, the thread gets the processor.

Not Runnable State: A thread that is not executable because

Sleep() method is called.

Wait() method is called.

Due to I/O request.

Suspend() method is called.

Dead State: When the thread completes its task, then thread enters into dead, terminates, abort state.

Thread class provides different types of methods to implement the states of the threads.

Sleep() method is used to temporarily suspend the current execution of the thread for specified milliseconds, so that other threads can get the chance to start the execution, or may get the CPU for execution.

Join() method is used to make all the calling thread to wait until the main thread, i.e. joined thread complete its work.

Abort() method is used to abort the thread.

Suspend() method is called to suspend the thread.

Resume() method is called to resume the suspended thread.

Start() method is used to send a thread into runnable State.

[Lifecycle and States of a Thread in C# - GeeksforGeeks](https://www.geeksforgeeks.org/lifecycle-and-states-of-a-thread-in-c-sharp/?ref=lbp)

Types of Thread

● In C# we can create two types of threads in the application, they are:

○ Foreground Thread

○ Background Thread

Foreground Thread vs. Background Thread

● Foreground threads are those threads that keep running even after the main

application exits or quits. So, the foreground threads do not care whether the

main thread is alive or not, it completes only when it finishes its assigned

work. That means the life of a foreground thread does not depend upon the

main thread. Foreground thread is the default type when a new thread is

created.

● Background Threads are those threads that will quit if our main thread is

finished. The life of a background thread depends on the main thread.

○ A thread can be changed to a background thread at any time by setting it’s IsBackground

property to true

1. How to create a thread in C#?

Thread Class

● In C#, a multi-threading system is built upon the Thread class, which

encapsulates the execution of threads.

● This class contains several methods and properties which helps in managing

and creating threads and this class is defined under System.Threading

namespace.

● The first thread to be executed in a process is called the main thread.

● When a C# program starts execution, the main thread is automatically

created. The threads created using the Thread class are called the child

threads of the main thread.

● Programmers can always take control of the main thread.

● The thread class provides lots of properties. Some of the important properties

are as follows:

● CurrentThread: used to get the current running thread.

● Name: used to get or set the name of the thread.

● Priority: used to get or set the priority value(Enum) of the thread.

● ThreadState: used to get the thread state value(Enum) of the thread.

● IsAlive: returns a bool value representing whether or not this thread is alive

● IsBackground: used to get or set the value(bool) indicating whether the

thread is a background thread or not.

Graphical user interface, text, application

Description automatically generated

Table

Description automatically generated

1. First of all import System.Threading namespace, it plays an important role in creating a thread in your program as you have no need to write the fully qualified name of class everytime.
2. Using System;

Using System.Threading

1. Now, create and initialize the thread object in your main method.

public static void main()

{

Thread thr = new Thread(job1);

}

**Or**

You can also use ThreadStart constructor for initializing a new instance.

public static void main()

{

Thread thr = new Thread(new ThreadStart(job1));

}

1. Now you can call your thread object.

public static void main()

{

Thread thr = new Thread(job1);

thr.Start();

}

[How to create Threads in C# - GeeksforGeeks](https://www.geeksforgeeks.org/how-to-create-threads-in-c-sharp/)

Text

Description automatically generated

[Threads In C# (c-sharpcorner.com)](https://www.c-sharpcorner.com/article/Threads-in-CSharp/#:~:text=Create%20and%20start%20a%20thread%20in%20C%23%201,3%20%2F%2F%20Start%20secondary%20thread%204%20workerThread.Start%20%28%29%3B)

There are a few different ways to create and start a new thread in C#. Here are two common approaches:

1. Using the **Thread** class: The **System.Threading.Thread** class provides static methods for creating and starting new threads. To create a new thread, you can use the **Thread** class's **Start()** method, which takes a **ThreadStart** delegate as an argument. The **ThreadStart** delegate is a function that represents the method that will be executed on the new thread. Here is an example of creating a new thread using the **Thread** class:

using System.Threading;

public static void Main()

{

Thread t = new Thread(new ThreadStart(ThreadMethod));

t.Start();

}

public static void ThreadMethod()

{

// Code to execute on the new thread

}

1. Using the **Task** class: The **System.Threading.Tasks.Task** class provides a more modern and flexible way of creating and starting new threads. To create a new thread, you can use the **Task** class's **Run()** method, which takes a **Action** delegate as an argument. The **Action** delegate is a function that represents the method that will be executed on the new thread. Here is an example of creating a new thread using the **Task** class:

using System.Threading.Tasks;

public static void Main()

{

Task t = Task.Run(() => ThreadMethod());

}

public static void ThreadMethod()

{

// Code to execute on the new thread

}

In both examples, the new thread is created and started when the **Start()** or **Run()** method is called. The code in the **ThreadMethod()** function will be executed on the new thread, while the main thread (the thread that called **Start()** or **Run()**) continues to execute concurrently.

1. What is data interference? How can we avoid data interference?

Thread Safety

● Threads communicate primarily by sharing access to the same recourses

such as fields or references to objects.

● This form of communication is extremely efficient, but it also makes some

problems possible: thread interference and data inconsistency.

Graphical user interface, application, Teams

Description automatically generated

Thread Interference

• Consider a situation where

two thread is operating on the

same object at the same time.

• Interference happens when

two operations, running in

different threads, but acting on

the same data, interleave. This

means that the two operations

consist of multiple steps, and

the sequences of steps overlap

The App will decompose the Increment method into following steps: • Retrieve the current Value. • Increment the retrieved value by 1. • Store the incremented value back in the Property. What if thread A is calling increment and thread B is calling decrement? (What will be the result?) This situation is also called the Race Condition

Data Inconsistency occurs

● Data Inconsistency when different threads have inconsistent views of what

should be the same data

● Eg. SameCounter class as before—Thread A increases the counter by 1, and

thread B tries to print the value of counter at the same time. Now the value

can either be 1 or 0

Use Thread Synchronization to resolve data inconsistency with lock keyword.

● Synchronization can be achieved by using the lock keyword.

● It is used lock the object and only allow one thread to access the locked

object, execute the task and then the lock will be released.

● It ensures that other thread does not interrupt the execution until the

execution finish.

● The lock can only apply on objects.

● Syntax:

Graphical user interface, text

Description automatically generated

Thread-Safe Collections can also resolve the data consistency

● The following table list some thread-safe collections under

System.Collections.Concurrent namespace: Graphical user interface, text, application

Description automatically generated

Data interference refers to the unintended modification or access of data by multiple threads or processes. It can occur when multiple threads or processes try to read from or write to the same data simultaneously, or when one thread or process modifies data that is being accessed by another thread or process.

There are a few ways to avoid data interference in C#:

1. Use synchronization primitives: C# provides a range of synchronization primitives, such as locks, semaphores, and monitors, that can be used to synchronize access to shared data. These primitives allow you to block access to shared data until it is no longer in use, which helps to prevent multiple threads or processes from accessing or modifying the data simultaneously.
2. Use the **volatile** keyword: The **volatile** keyword can be used to declare a field as volatile, which tells the compiler to generate code that ensures that the field is accessed in a thread-safe manner. This can be useful for fields that are accessed by multiple threads or processes, as it helps to prevent data interference by ensuring that the field is always accessed in a consistent and predictable way.
3. Use the **Interlocked** class: The **System.Threading.Interlocked** class provides a range of atomic operations that can be used to safely update shared variables from multiple threads or processes. These operations allow you to perform operations on shared variables in a thread-safe manner, without the need for locks or other synchronization primitives.

By using these techniques, you can help to avoid data interference and ensure that shared data is accessed and modified in a consistent and predictable way.

1. What is a deadlock? How to avoid deadlock?

Deadlocks occur when you have two threads that have a lock on resources the other thread needs. Since the threads are mutually waiting on the other thread to release the lock, neither will ever complete.

The best way to deal with deadlocks is to never lock in the first place.

 To prevent race conditions (the opposite of a deadlock), you can use collections that are designed for concurrent use like ConcurrentQueue or almost everything the [System.Collections.Concurrent](https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent?view=netframework-4.8) namespace.

If that's not possible, the following rules of thumb will at least minimize the likelihood of deadlocks if they don't completely eliminate them:

* Keep the period of the lock as short as possible
* Avoid acquiring multiple locks in the same thread
* Release locks before delegating tasks to background threads (including tasks)
* At the very least, locks should be released in the reverse order that they were acquired

Also know the impacts of the locking mechanisms you choose:

* lock keyword and code block: light weight lock across threads that is released as soon as the code exits the code block
* Semephore: light weight semaphore that locks across threads, same mechanics as a mutex though
* Mutex: very heavy weight mutex that protects resources across processes, uses the file system to manage the lock. Can be used to ensure only one instance of an application is ever running at a time.

The lock keyword is pretty straightforward and the easiest to get right. The Mutex has many different error states that you have to make doubly sure that you have the lock or not.

[c# - How to avoid deadlock - Software Engineering Stack Exchange](https://softwareengineering.stackexchange.com/questions/403248/how-to-avoid-deadlock#:~:text=Deadlocks%20occur%20when%20you%20have%20two%20threads%20that,is%20to%20never%20lock%20in%20the%20first%20place.)

[Avoiding And Detecting Deadlocks In .NET Apps with C# and C++ | Microsoft Learn](https://learn.microsoft.com/en-us/archive/msdn-magazine/2006/april/avoiding-and-detecting-deadlocks-in-net-apps-with-csharp-and-c)

● Deadlock describes a situation where two or more threads are blocked

forever, waiting for each other to release the lock

Diagram

Description automatically generated

A deadlock is a situation in which two or more threads are blocked and waiting for each other to release a resource, resulting in a standstill. Deadlocks can occur when multiple threads attempt to acquire locks on resources in a different order, or when a thread attempts to acquire a lock that it already holds.

Deadlocks can be difficult to detect and resolve, as they can occur intermittently and may not always be reproducible. They can also lead to serious performance issues, as they can cause threads to become blocked and unresponsive.

To avoid deadlocks, you can follow these best practices:

1. Avoid nested locks: Nested locks can lead to deadlocks, as they can cause threads to acquire locks in different orders. Instead of using nested locks, try to use a single lock to synchronize access to shared resources.
2. Use lock timeouts: Lock timeouts allow you to specify a maximum amount of time that a thread should wait for a lock to be released. If the lock is not released within the specified time, the thread can continue execution without the lock. This can help to prevent deadlocks by allowing threads to move on if a lock is not available.
3. Use lock hierarchies: Lock hierarchies allow you to specify an order in which locks should be acquired, which can help to prevent deadlocks. For example, you might define a hierarchy in which locks on resources A and B are always acquired in the same order.

# 10. What’s the difference between IEnumerable and IQueryable?

# IQueryable

# ● The IQueryable interface inherits the IEnumerable interface so that if it

# represents a query, the results of that query can be enumerated.

# ● We can use AsQueryable() and AsEnumerable() to convert them

# Diagram Description automatically generated

**IEnumerable<T>** and **IQueryable<T>** are interfaces in the .NET framework that represent a sequence of elements that can be enumerated. Both interfaces provide methods for iterating over the elements in the sequence, such as **GetEnumerator** and **MoveNext**.

There are several differences between **IEnumerable<T>** and **IQueryable<T>**, the most important of which is that **IQueryable<T>** represents a query that can be executed, whereas **IEnumerable<T>** represents a sequence of elements that has already been materialized.

Here are some key differences between **IEnumerable<T>** and **IQueryable<T>**:

* **IEnumerable<T>** is intended for in-memory collections, whereas **IQueryable<T>** is intended for external data sources such as databases.
* **IEnumerable<T>** represents a sequence of elements that has already been materialized, whereas **IQueryable<T>** represents a query that can be executed.
* **IEnumerable<T>** is implemented by classes that hold the data in memory, such as **List<T>**, whereas **IQueryable<T>** is implemented by classes that represent a query against a data source, such as **DbSet<T>** in Entity Framework.
* **IEnumerable<T>** provides methods for iterating over the elements in the sequence, such as **GetEnumerator** and **MoveNext**, whereas **IQueryable<T>** provides methods for executing a query, such as **Provider.Execute<TResult>(Expression)**.

# Coding Questions:

Write code in c# to solve the following problems. Please write your own answers. You are highly encouraged to present more than one way to answer the questions. Please follow best practices when you write the code so that it is easily readable, maintainable, and efficient. Clearly state your assumptions if you have any. You may discuss with others on the questions, but please write your own code.

## \*Please create the corresponding type

1. Given a list of int

int[] arr1 = new int[]{3,9,2,8,6,5, 3, 2, 2};

## Using LINQ, Find the number and its square of an array which is more than 20

The output should look like this:

{ Number = 9, SqrNo = 81 }

{ Number = 8, SqrNo = 64 }

{ Number = 6, SqrNo = 36 }

{ Number = 5, SqrNo = 25 }

## Using LINQ, display the number and frequency of number from given array

The output should look like this:

Number 2 appears 3 times

Number 3 appears 2 times

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Number | 9 | appears | 1 | times |
| Number | 8 | appears | 1 | times |
| Number | 6 | appears | 1 | times |
| Number | 5 | appears | 1 | times |

1. Given a string of words

Eg. string sentence = "the quick brown fox jumps over the lazy dog";

Using LINQ, Group the words by their length in ascending order and display them. The output should look like this:

Words of length 3:

THE, FOX, THE, DOG

Words of length 4:

OVER, LAZY

Words of length 5:

QUICK, BROWN, JUMPS

\*use both query syntax and method syntax to achieve that

1. Given a Car list

var c = new Car[]

{

new Car{ Color="Blue", Price=28000}, new Car{ Color="Red", Price=54000}, new Car{ Color="Pink", Price=9999},

new Car{ Color="Black", Price=46722}, new Car{ Color="White", Price=35264}

};

* 1. Find the index of the first car whose Price is smaller than 10000 with LINQ?
  2. Find the colors for the top 3 highest Car price

1. Generate your own list of students and teacher

class Student

{

public string First { get; set; } public string Last {get; set;}

public int ID { get; set; }

public string Street { get; set; } public string City { get; set; }

public List<int> Scores;

}

class Teacher

{

public string First { get; set; } public string Last { get; set; } public int ID { get; set; }

public string City { get; set; }

}

* 1. Using LINQ, display the student name and the teacher name who live in the same city
  2. Using LINQ, generate a list of objects, each object contains the **student name** and the **teacher name** who live in the same city and the city name is “New York”