**TASK 1 - .NET FUNDAMENTALS**

* **What does the .NET ecosystem provide?**

The .NET ecosystem is a collection of tools, libraries, and frameworks that are used to develop, deploy, and run applications on the .NET platform. The ecosystem includes the following components:

1. **.NET Framework**: A software framework developed by Microsoft that runs primarily on Microsoft Windows operating systems. It includes a large class library known as Framework Class Library (FCL) and provides language interoperability across several programming languages.
2. **.NET Core**: A free, open-source, cross-platform framework for building modern cloud-based applications. It is designed to be modular and lightweight and can be used to build applications for Windows, Linux, and macOS.
3. **ASP.NET**: A web application framework developed by Microsoft that allows developers to build dynamic web sites, web applications, and web services.
4. **Entity Framework**: An object-relational mapping (ORM) framework that allows developers to work with relational data using domain-specific objects.
5. **Visual Studio**: An integrated development environment (IDE) used to develop computer programs, websites, web apps, mobile apps, and cloud services.
6. **NuGet**: A package manager for the Microsoft development platform including .NET. It is used to install, update, and manage packages in a project.
7. **Roslyn Compiler**: A set of open-source compilers and code analysis APIs for C# and Visual Basic .NET languages.
8. **Xamarin**: A cross-platform app development platform that allows developers to create native iOS, Android, and Windows apps using a single codebase.
9. **Azure**: A cloud computing service created by Microsoft for building, testing, deploying, and managing applications and services through Microsoft-managed data centers.

* **What are .NET implementations? Which ones are used nowadays?**
* **.NET Framework**: A software framework developed by Microsoft that runs primarily on Microsoft Windows operating systems. It includes a large class library known as Framework Class Library (FCL) and provides language interoperability across several programming languages.
* **.NET Core**: A free, open-source, cross-platform framework for building modern cloud-based applications. It is designed to be modular and lightweight and can be used to build applications for Windows, Linux, and macOS.
* **Mono**: A cross-platform implementation of the .NET Framework that is mainly used when a small runtime is required. It supports all of the currently published .NET Standard versions .

Microsoft supports four .NET implementations: .NET 5 (and .NET Core) and later versions, .NET Framework, Mono, and Universal Windows Platform (UWP). The latest version of .NET is .NET 7, which is built on a single code base that supports multiple platforms and many workloads such as Windows desktop apps and cross-platform console apps, cloud services, and websites.

Each implementation of .NET includes one or more runtimes, a class library, and optionally one or more application frameworks. The choice of implementation depends on the specific requirements of the application being developed.

* **What is CLR?**

The **Common Language Runtime (CLR)** is a component of the Microsoft .NET Framework that manages the execution of .NET applications. It is responsible for loading and executing the code written in various .NET programming languages, including C#, VB.NET, F#, and others. When a C# program is compiled, the resulting executable code is in an intermediate language called Common Intermediate Language (CIL) or Microsoft Intermediate Language (MSIL). This code is not machine-specific, and it can run on any platform that has the CLR installed. When the CIL code is executed, the CLR compiles it into machine code that can be executed by the processor. The CLR provides many services to .NET applications, including memory management, type safety, security, and exception handling. It also provides Just-In-Time (JIT) compilation, which compiles the CIL code into machine code on the fly as the program runs, optimizing performance. Additionally, the CLR provides a framework for developing and deploying .NET applications, including a set of libraries, called the .NET Framework Class Library, which provides access to a wide range of functionality, such as input/output operations, networking, database connectivity, and user interface design.

* **Why is .NET 5 a bit of a special version?**

.NET 5 is a bit of a special version for several reasons. Firstly, it is the **first release** of .NET that does not include the word “Core” in its name. Secondly, it is the **first release** of .NET that supports more types of apps and more platforms than .NET Core or .NET Framework. Thirdly, it is the **first release** of .NET that combines and replaces the netcoreapp and netstandard Target Framework Monikers (TFMs). Fourthly, it is the **first release** of .NET that includes many new features and improvements compared to .NET Core 3.1. Lastly, it is the **first release** of .NET that is built on a single code base that supports multiple platforms and many workloads such as Windows desktop apps and cross-platform console apps, cloud services, and websites.

* **Which technologies are supported by the .NET framework?**

The .NET framework is a collection of tools, libraries, and frameworks that are used to develop, deploy, and run applications on the .NET platform. The following are some of the technologies supported by the .NET framework:

1. **ASP.NET**: A web application framework developed by Microsoft that allows developers to build dynamic web sites, web applications, and web services.
2. **Windows Communication Foundation (WCF)**: A framework for building service-oriented applications.
3. **Windows Presentation Foundation (WPF)**: A graphical subsystem for rendering user interfaces in Windows-based applications.
4. **Windows Workflow Foundation (WF)**: A framework for building workflow-enabled applications.
5. **Windows CardSpace**: A software component that provides a consistent way for users to manage their digital identities.
6. **ADO.NET**: A set of classes that expose data access services for .NET Framework programmers.
7. **LINQ**: A set of extensions to the .NET Framework that provides a standard way to query data from different data sources.
8. **Parallel Extensions**: A managed concurrency library that simplifies the development of concurrent and parallel applications.
9. **Managed Extensibility Framework (MEF)**: A framework for building extensible applications.
10. **Windows Azure**: A cloud computing platform developed by Microsoft for building, deploying, and managing applications and services through a global network of Microsoft-managed data centers.

* **Does cross-platform development is possible in .NET? What about cross-platform UI?**

Yes, cross-platform development is possible in .NET. **.NET Core** is a free, open-source, cross-platform framework for building modern cloud-based applications that can be used to build applications for Windows, Linux, and macOS. Additionally, .NET Multi-platform App UI (**MAUI**) is a cross-platform framework for creating native mobile and desktop apps with C# and XAML that can run on Android, iOS, macOS, and Windows from a single shared code-base.

* **What does the multitargeting term mean?**

In the context of .NET, multitargeting refers to the ability to compile applications that can run on different versions of the .NET runtime. It allows developers to target multiple versions of the .NET Framework or .NET Core with a single codebase. Multitargeting ensures that your application does not use any types or methods that were introduced in newer versions of the runtime than the one you are targeting.

**TASK 2 - DEBUGGING FUNDAMENTALS**

* **What is break mode? What are the options to step through code?**

Break mode in .NET is a debugging mode that allows you to observe code line by line in order to locate errors. When changes are made to the code in an application, the way to be able to view how those changes have changed the way of execution is Break Mode. In break mode, a snapshot of the running application is taken in which the status and values of all the variables is stored. Visual Studio provides following options to step through code:

1. **Step Into**: This option steps into the next statement, whether it is a method call or a property access.
2. **Step Over**: This option executes the next statement and then returns control to the current line of code.
3. **Step Out**: This option executes the remaining statements in the current method and then returns control to the calling method.
4. **Run To Cursor**: This option runs the application until it reaches the cursor position.
5. **Set Next Statement**: This option allows you to change the execution point of your code.

* **How to ignore some exceptions during debugging?**

To ignore exceptions during debugging in Visual Studio, you can use the Exception Settings window. In this window, you can specify on which exceptions to break. If you want to ignore a specific exception, you can uncheck the box next to it.

* **How to set up conditional breakpoint?**

To set up a conditional breakpoint in Visual Studio, you can follow these steps:

1. Set a breakpoint at the line of code where you want to break execution.
2. Right-click on the breakpoint and select “Condition” from the context menu.
3. In the “Breakpoint Condition” dialog box, enter the condition that you want to use to trigger the breakpoint.
4. Click “OK” to save the condition.

Now, when the code reaches that line during debugging, it will only break if the condition is true.

* **What is data breakpoint?**

A data breakpoint is a type of breakpoint that allows you to break execution when the value stored at a specified memory location changes. This can be useful when you want to monitor changes to a variable’s value or track down memory-related issues in your code.

* **What is trace point and how to use it?**

A tracepoint is a debugging feature in Visual Studio that allows you to log information to the Output window under configurable conditions without modifying or stopping your code. Tracepoints are supported for both managed languages (C#, Visual Basic, F#) and native code as well as languages such as JavaScript and Python.

To use tracepoints in Visual Studio, you can follow these steps:

1. Set a breakpoint at the line of code where you want to log information.
2. Right-click on the breakpoint and select “When Hit…” from the context menu.
3. In the “When Hit” dialog box, enter the message that you want to log into the “Print a message” text box.
4. Click “OK” to save the tracepoint.

Now, when the code reaches that line during debugging, it will log the message you specified to the Output window.

* **What are pdb files, when are they created and how to use them?**

A **program database (PDB) file** is a binary file that contains debugging and project state information that allows incremental linking of a Debug configuration of your program. PDB files are created when you compile a Visual Basic/C#/.NET program with /debug.

PDB files hold debugging and project state information that allows incremental linking of a Debug configuration of your program. They contain type and symbolic debugging information gathered over the course of compiling and linking the project. PDB files are as important as source code, as they help link crash reports back to the source code.

To use PDB files in Visual Studio, you can follow these steps:

1. Build your project with the /debug option.
2. In the “Solution Explorer” window, right-click on your project and select “Properties”.
3. In the “Properties” window, select “Linker” -> “Debugging”.
4. Set the “Generate Debug Info” option to “Yes (/DEBUG)”.
5. Set the “Debug Info Format” option to “Program Database (/Zi)”.
6. Click “OK” to save the changes.

Now, when you run your program in Debug mode, Visual Studio will use the PDB file to provide debugging information.

* **What is symbol server?**

The **Microsoft symbol server** is a debugging feature in Visual Studio that allows you to download debugging symbols for Windows operating systems and applications. Debugging symbols contain information about the executable code of a program, such as function names, variable names, and line numbers. This information is used by the debugger to provide more detailed information about the program’s execution.

* **What are debug and release build configurations?**

In Visual Studio, **debug** and **release** configurations are two different build configurations that can be used to compile your code.

The **debug configuration** is optimized for debugging and is used to compile your code with full symbolic debug information and no optimization. This makes it easier to debug your code, but it also makes the executable file larger and slower.

The **release configuration** is optimized for performance and is used to compile your code with no symbolic debug information and full optimization. This makes the executable file smaller and faster, but it also makes it harder to debug your code.

**TASK 3 – ADVANCED C#**

* **What is the difference between reference and value types?**

In C#, data types are categorized based on how they store their value in the memory. Value types hold a data value within their own memory space, while reference types store the address where the value is being stored.

Value types include all numeric data types, Boolean, Char, Date, and structures, even if their members are reference types. When you pass a value-type variable from one method to another, the system creates a separate copy of the variable in another method. If the value got changed in one method, it wouldn’t affect the variable in another method.

Reference types include String, Arrays (even if their elements are value types), Class, and Delegate. Unlike value types, a reference type doesn’t store its value directly. Instead, it stores the address where the value is being stored. In other words, a reference type contains a pointer to another memory location that holds the data. When you pass a reference type variable from one method to another, it doesn’t create a new copy; instead, it passes the variable’s address. So if we change the value of a variable in a method, it will also be reflected in the calling method.

* **What is boxing and unboxing?**

In C#, **boxing** is the process of converting a value type to the type object or to any interface type implemented by this value type. When the common language runtime (CLR) boxes a value type, it wraps the value inside a System.Object instance and stores it on the managed heap. **Unboxing** extracts the value type from the object. Boxing is implicit; unboxing is explicit.

The concept of boxing and unboxing underlies the C# unified view of the type system in which a value of any type can be treated as an object.

* **What is a class? What is an interface?**

In C#, a **class** is a user-defined type that acts as a blueprint for object creation. It can have properties and methods that represent the states and behaviors of an object, respectively. A class can be instantiated, i.e., objects of a class can be created.

An **interface** is also a user-defined type that is syntactically similar to a class. However, it only specifies what a class must do and not how. It is the blueprint of the class. An Interface cannot be instantiated, i.e., objects cannot be created.

1. A class is a blueprint for creating objects, whereas an interface is a contract that defines the behavior of a class.
2. A class can be instantiated, i.e., objects of a class can be created. An interface cannot be instantiated, i.e., objects cannot be created.
3. A class can have constructors and destructors. An interface cannot have constructors or destructors.
4. A class can have fields, properties, methods, events, and indexers. An interface can only have methods, properties, events, and indexers.
5. A class can support inheritance. An interface supports multiple inheritance.
6. A class can implement an interface. An interface cannot implement a class.

In general, classes are used to define objects that have state and behavior, while interfaces are used to define the behavior of objects.

* **What is the difference between class and structure?**

In C#, both **class** and **structure** are user-defined types that can be used to define objects. However, there are some differences between them.

In general, classes are used for more complex data structures that require inheritance or other advanced features, while structures are used for simple data structures that do not require inheritance or other advanced features.

1. A class is a reference type, whereas a structure is a value type.
2. A class can be inherited from another class, whereas a structure cannot be inherited from another structure or class.
3. A class can have a constructor and a destructor, whereas a structure can have only a constructor.
4. A class can have fields, properties, methods, events, and indexers, whereas a structure can have fields, properties, methods, and indexers, but not events.
5. A class can have both static and instance members, whereas a structure can have only instance members.

* **What is a generic type? What is Covariance and Contravariance?**

In C#, a **generic type** is a type that can be parameterized with one or more types. It allows you to define a class, structure, interface, or method that can work with any data type.

**Covariance** and **Contravariance** are terms that refer to the ability to use a more derived type (more specific) or a less derived type (less specific) than originally specified. Generic type parameters support covariance and contravariance to provide greater flexibility in assigning and using generic types.

* **Covariance** enables you to use a more derived type than originally specified. You can assign an instance of IEnumerable<Derived> to a variable of type IEnumerable<Base>.
* **Contravariance** enables you to use a more generic (less derived) type than originally specified. You can assign an instance of Action<Base> to a variable of type Action<Derived>.
* **Invariance** means that you can use only the type originally specified. An invariant generic type parameter is neither covariant nor contravariant. You cannot assign an instance of List<Base> to a variable of type List<Derived> or vice versa.
* **What is delegate?**

In C#, a **delegate** is a type that represents a reference to a method with a specific signature. It is similar to a function pointer in C or C++. A delegate can be used to pass methods as arguments to other methods, or to store methods as instance variables.

* **What is event?**

In programming, an **event** is an action or occurrence recognized by software, often originating asynchronously from the external environment, that may be handled by the software. In event-driven programming, the flow of the program is determined by events such as user actions from mice, keyboards, touchpads and touchscreens. Non-user initiated events can involve sensor inputs or be programmatically generated (message passing) from other programs or threads.

Event-driven programming is the dominant paradigm used in graphical user interfaces and other applications (e.g., JavaScript web applications) that are centered on performing certain actions in response to user input.

* **What is the difference between delegate and event?**

1. A delegate is a function pointer, whereas an event is a notification mechanism that depends on delegates.
2. A delegate is independent and not dependent on events, whereas an event is dependent on a delegate and cannot be created without delegates.
3. An event includes AddEventHandler() and RemoveEventHandler() methods to add and remove methods from the invocation list, respectively, whereas a delegate includes Combine() and Remove() methods to add methods to the invocation list.
4. A delegate can be passed as a method parameter, whereas an event cannot be passed as a method parameter.
5. The = operator is used to assign a single method, and the += operator is used to assign multiple methods to a delegate. The = operator cannot be used with events, and only the += and -= operators can be used with an event that adds or removes an event handler.

**TASK 4 – EXCEPTION HANDLING**

* **What is the purpose of Exception Handling in C#?**

Exception handling is a mechanism in C# that allows developers to handle runtime errors and exceptions that occur during program execution. It is implemented using the **try**, **catch**, and **finally** keywords.

The purpose of exception handling is to provide a way to recover from errors and prevent the program from crashing. When an exception occurs, the program can gracefully handle the error and continue executing, rather than abruptly terminating.

The **try** block contains the code that might throw an exception, while the **catch** block contains the code that handles the exception. The **finally** block contains code that is executed regardless of whether an exception is thrown or not.

By using exception handling, developers can write more robust and reliable code that can handle unexpected situations. It also helps in debugging and troubleshooting by providing detailed information about the error that occurred.

* **Can a try block have multiple catch blocks?**

Yes, a **try** block can have multiple **catch** blocks in C#. Each **catch** block can handle a different type of exception. The order of the **catch** blocks is important, and they should be ordered from most specific to most general. If an exception is thrown in the **try** block, the runtime searches for the first **catch** block that can handle the exception and executes it. If no matching **catch**block is found, the exception is propagated up the call stack.

* **Describe a flow how exceptions are handled?**

In C#, when an exception occurs in the **try** block, the flow of control jumps to the first associated **catch** block that is present anywhere in the call stack. If no **catch** block for a given exception is present, the program stops executing with an error message.

The **catch** keyword is used to define an exception handler. A **try** block can have multiple **catch** blocks in C#. Each **catch** block can handle a different type of exception. The order of the **catch** blocks is important, and they should be ordered from most specific to most general.

If an exception is thrown in the **try** block, the runtime searches for the first **catch** block that can handle the exception and executes it. If no matching **catch** block is found, the exception is propagated up the call stack.

The **finally** block contains code that is executed regardless of whether an exception is thrown or not.

* **What is the base class from which all exceptions are derived?**

In C#, all the exceptions are derived from the base class **System**.**Exception**. The **SystemException** class is the base class for all predefined system exceptions, while the **ApplicationException** class is the base class for all application-related exceptions.

* **What is the difference between Exception and Inner Exception?**

In C#, **Exception** is the base class for all exceptions, while **Inner Exception** is a property of the **Exception** class that contains the exception that caused the current exception.

The **InnerException** property returns the same value as was passed into the **Exception (String, Exception)** constructor, or null if the inner exception value was not supplied to the constructor.

A program can catch an exception and re-raise a different exception, passing the original caught exception as the **InnerException**. The **Exception (String, Exception)** constructor does this, for example. This happens in the .NET FW itself, TypeInitializationException, TypeLoadException, TargetInvocationException, etc., are raised this way. The inner exception is completely unrelated to the raised exception and it is very important that you log the inner exception as well to have any hope of diagnosing the root cause of the problem.

* **What is the difference between throw ex; and throw; statements?**

In C#, **throw** and **throw ex;** are two different statements used for exception handling.

When an exception is thrown using the **throw** statement, a new exception object is created, which can be an expensive operation in terms of performance. On the other hand, when an exception is rethrown using the **throw ex;** statement, no new exception object is created, which can result in better performance.

The difference between **throw** and **throw** **ex;** is that **throw** preserves the original stack trace information of the exception. In contrast, **throw ex;** resets the stack trace of the exception to the current method, which can make it difficult to diagnose the root cause of the problem.

* **What is the purpose of finally statement?**

The **finally** block is executed after the **try** and **catch** blocks, even if an exception is thrown. This makes it useful for releasing resources, such as closing files or database connections, that were opened in the **try** block.

* **What predefined .NET Exceptions do you know?**

In C#, there are several predefined exception classes that are derived from the **System.Exception** base class. Here are some of the most commonly used ones:

1. **System**.**ArgumentException**: Thrown when an argument is invalid.
2. **System**.**ArgumentNullException**: Thrown when a null argument is passed to a method that does not accept it.
3. **System**.**ArgumentOutOfRangeException**: Thrown when an argument is outside the range of acceptable values.
4. **System**.**DivideByZeroException**: Thrown when an attempt is made to divide by zero.
5. **System**.**FormatException**: Thrown when a string cannot be parsed into the expected format.
6. **System**.**IndexOutOfRangeException**: Thrown when an index is outside the bounds of an array or collection.
7. **System**.**InvalidCastException**: Thrown when an invalid cast is attempted.
8. **System**.**InvalidOperationException**: Thrown when an operation is not valid in the current state.
9. **System**.**IO**.**IOException**: Thrown when an I/O error occurs.
10. **System**.**NotImplementedException**: Thrown when a method or operation is not implemented.

* **Is there a way to create a custom exception?**

Yes, you can create a custom exception in C# by creating a new class that derives from the **System**.**Exception** base class.

You can also add additional properties or methods to your custom exception class as needed.

**TASK 5 – REFLECTION**

* **What is reflection in .NET?**

Reflection is a powerful feature of .NET that allows you to obtain information about types, methods, properties, and fields at runtime. It is implemented through the classes in the **System.Reflection** namespace, which enable you to obtain information about loaded assemblies and the types defined within them. You can also use reflection to create type instances at run time, and to invoke and access them. Reflection is useful in situations such as when you have to access attributes in your program’s metadata, examine and instantiate types in an assembly, or build new types at run time.

* **What does reflection allow you to do?**

Reflection allows you to obtain information about types, methods, properties, and fields at runtime. It is useful in situations such as when you have to access attributes in your program’s metadata, examine and instantiate types in an assembly, or build new types at run time.

* **What are fully qualified type names?**

In .NET reflection, fully qualified type names are used to specify the type of an object. A fully qualified type name consists of an assembly name specification, a namespace specification, and a type name. You can use fully qualified type names to have valid input to various reflection operations such as **Type.GetType**, **Module**.**GetType**, **ModuleBuilder**.**GetType**, and **Assembly.GetType**.

* **What examples of practical application of reflection can you imagine?**

1. **Dependency Injection**: Reflection can be used to create instances of classes dynamically, which is useful for dependency injection frameworks. Dependency injection frameworks use reflection to automatically inject dependencies into objects at runtime.
2. **Serialization**: Reflection can be used to serialize and deserialize objects. Serialization is the process of converting an object into a format that can be stored or transmitted, while deserialization is the process of converting the serialized data back into an object.
3. **Dynamic Loading**: Reflection can be used to load assemblies and types dynamically at runtime. This is useful when you need to load types that are not known at compile time.
4. **Attribute-Based Programming**: Reflection can be used to read attributes from code at runtime. Attributes are used to add metadata to code, such as information about how it should be serialized or how it should be displayed in a user interface.
5. **Code Generation**: Reflection can be used to generate code dynamically at runtime. This is useful when you need to create code that is not known at compile time, such as proxy classes for web services.

* **Is it possible to get information about private fields/methods using reflection?**

Yes, it is possible to access private fields and methods using reflection in .NET. You can use the **Type.GetField**and **Type.GetFields** methods to get information about private fields, and the **Type.GetMethod**and **Type.GetMethods**methods to get information about private methods.

You can also use the **BindingFlags.NonPublic** flag to include non-public members (private, internal, and protected members) in the search.

However, accessing private members using reflection requires elevated permissions and is not recommended because it can lead to unexpected behavior and security issues.

**TASK 6 – Data Structures**

* **What is a Data Structure?**

A **data structure** is a way of organizing and storing data in a computer so that it can be accessed and updated efficiently. It is not only used for organizing the data, but also for processing, retrieving, and storing data. There are different basic and advanced types of data structures that are used in almost every program or software system that has been developed.

A **linear data structure** is a data structure in which data elements are arranged sequentially or linearly, where each element is attached to its previous and next adjacent elements. Examples of linear data structures are array, stack, queue, linked list, etc. A **non-linear data structure** is a data structure where data elements are not placed sequentially or linearly. Examples of non-linear data structures are trees and graphs.

Data structures can be classified into four categories: **linear**, **non-linear**, **static**, and **dynamic**. A static data structure has a fixed memory size, while a dynamic data structure has a variable memory size that can be updated during runtime.

* **What are the various operations that can be performed on different Data Structures?**

**Traversing**: Traversing a data structure means to visit the element stored in it. It visits data in a systematic manner. This can be done with any type of data structure.

**Searching**: Searching means to find a particular element in the given data structure. It is considered successful when the required element is found. Searching is the operation which we can perform on data structures like array, linked-list, tree, graph, etc.

**Insertion**: Insertion means to add an element in the given data structure. The operation of insertion is successful when the required element is added to the required data structure.

**Deletion**: Deletion means to remove an element from the given data structure. The operation of deletion is successful when the required element is removed from the required data structure.

**Sorting**: Sorting means arranging elements in a particular order. It can be ascending or descending order.

**Merging**: Merging means combining two or more data structures into one.

**Splitting**: Splitting means dividing a single data structure into two or more parts.

* **How is an Array different from Linked List?**

An **array** is a collection of elements of similar data types that are stored in contiguous memory locations. The elements can be accessed using an index, which represents the position of the element in the array. Arrays have a fixed size, which means that the size of an array is determined at the time of declaration and cannot be changed during runtime.

On the other hand, a **linked list** is a collection of nodes that are connected to each other through pointers. Each node contains two fields: one field stores the data, and the other field stores the address of the next node. Linked lists do not have a fixed size, which means that they can grow or shrink during runtime.

The main difference between an array and a linked list is that arrays are static data structures, while linked lists are dynamic data structures. This means that arrays have a fixed size, while linked lists can grow or shrink during runtime. In addition, accessing an element in an array is faster than accessing an element in a linked list because arrays store elements in contiguous memory locations, while linked lists store elements randomly in memory locations.

* **What is Stack and where can it be used?**

A **stack** is a linear data structure that follows a particular order in which the operations are performed. The order may be LIFO (Last In First Out) or FILO (First In Last Out). LIFO implies that the element that is inserted last comes out first, and FILO implies that the element that is inserted first comes out last.

Stacks are used in many applications, including:

1. **Undo/Redo functionality**: Stacks are used to enable undo and redo operations in various applications like text editors, graphic design tools, and software development environments.
2. **Backtracking**: Stacks can be used for backtracking, i.e., to check parenthesis matching in an expression.
3. **Expression conversion and evaluation**: A stack is a very effective data structure for evaluating arithmetic expressions in programming languages.
4. **Memory management**: Stacks can be used for systematic memory management.

* **What is a Queue, how it is different from the stack and how is it implemented?**

A **queue** is a linear data structure that follows the **FIFO (First In First Out)** principle, i.e., the element inserted first is the first element to be removed from the list. Elements can be inserted only from one side of the list called rear, and the elements can be deleted only from the other side called front. The insertion of an element in a queue is called an enqueue operation, and the deletion of an element is called a dequeue operation.

The main difference between a stack and a queue is that stacks are based on the LIFO principle, while queues are based on the FIFO principle. Stacks are often used for tasks that require backtracking, such as parsing expressions or implementing undo functionality. Queues are often used for tasks that involve processing elements in a specific order, such as handling requests or scheduling tasks.

Stacks are implemented using an array or linked list data structure. Queues are also implemented using an array or linked list data structure.

* **What is a Linked List and what are its types?**

A **linked list** is a linear data structure in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers. In simple words, a linked list consists of nodes where each node contains a data field and a reference (link) to the next node in the list.

There are different types of linked lists, some of which are:

**Singly Linked List**: It is the simplest type of linked list in which every node contains some data and a pointer to the next node of the same data type. A single linked list allows the traversal of data only in one way.

**Doubly Linked List**: A doubly linked list or a two-way linked list is a more complex type of linked list that contains a pointer to the next as well as the previous node in sequence. Therefore, it contains three parts of data: a pointer to the next node, a pointer to the previous node, and some data. This would enable us to traverse the list in both forward and backward directions.

**Circular Linked List**: A circular linked list is a type of linked list where the last node points back to the first node. This means that the last node of the list points to the first node, and the first node has a pointer to the last node.

**Doubly Circular Linked List**: A doubly circular linked list is a combination of both doubly linked and circular linked lists. It has two pointers, one pointing to the next node and another pointing to the previous node. The last node points back to the first node, and the first node points to the last node.

* **Which data structures C# have?**

C# provides a wide range of data structures that can be used to store and manipulate data. Some of the commonly used data structures in C# are:

**Array**: An array is a collection of elements of the same type that are stored in contiguous memory locations. The elements can be accessed using an index, which represents the position of the element in the array. Arrays have a fixed size, which means that the size of an array is determined at the time of declaration and cannot be changed during runtime.

**HashSet**: A hashset is a collection of unique elements where each element is unique. It provides fast access to elements based on their hash code.

**List**: A list is a dynamic data structure that can grow or shrink during runtime. It is similar to an array, but it has some additional features such as the ability to add or remove elements at any position in the list.

**Stack**: A stack is a linear data structure that follows a particular order in which the operations are performed. The order may be LIFO (Last In First Out) or FILO (First In Last Out). Stacks are often used for tasks that require backtracking, such as parsing expressions or implementing undo functionality.

**Queue**: A queue is a linear data structure that follows the FIFO (First In First Out) principle, i.e., the element inserted first is the first element to be removed from the list. Queues are often used for tasks that involve processing elements in a specific order, such as handling requests or scheduling tasks.

**Dictionary**: A dictionary is a collection of key-value pairs where each key is unique. It provides fast access to values based on their keys.

* **List C# collections features.**

In C#, there are two types of collections: **generic** and **non-generic**.

A **generic collection** is a collection that provides type safety without having to derive from a base collection type and implement type-specific members. It is a class that can store elements of a specific data type, and it provides better performance than non-generic collections.

Some of the most commonly used generic collections in C# are:

1. **List<T>**: A dynamic collection of elements of type T that can be accessed by index. It provides methods to add, remove, and search for elements.
2. **Dictionary<TKey, TValue>**: A collection of key-value pairs where each key is unique. It provides fast access to values based on their keys.
3. **HashSet<T>**: A collection of unique elements where each element is unique. It provides fast access to elements based on their hash code.
4. **Queue<T>**: A collection of elements with First-In-First-Out (FIFO) order. It keeps the order in which the values were added. It provides an Enqueue() method to add values and a Dequeue() method to retrieve values from the collection.
5. **Stack<T>**: A collection of elements with Last-In-First-Out (LIFO) order. It provides a Push() method to add a value and Pop() & Peek() methods to retrieve values.

A **non-generic collection** is a specialized class for data storage and retrieval that provides support for stacks, queues, lists, and hashtables. It can store elements of different data types, but it requires type casting to retrieve the elements.

Some of the most commonly used non-generic collections in C# are:

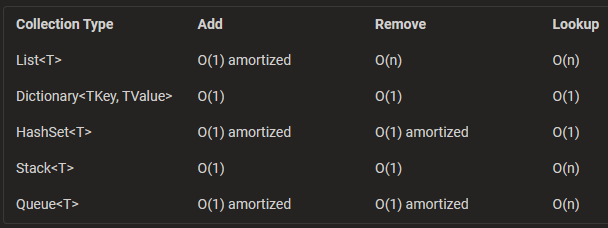
1. **ArrayList**: An ArrayList is a dynamic array of objects that can be indexed, sorted, and searched. ArrayList automatically resizes as elements are added or removed, making it an efficient collection for changing data sets.
2. **Hashtable**: A Hashtable is a collection of key-value pairs where each key is unique. It retrieves the values by comparing the hash value of the keys.
3. **Stack**: A Stack stores the values in LIFO style (Last In First Out). It provides a Push() method to add a value and Pop() & Peek() methods to retrieve values.
4. **Queue**: A Queue stores the values in FIFO style (First In First Out). It keeps the order in which the values were added. It provides an Enqueue() method to add values and a Dequeue() method to retrieve values from the collection.

* **What is the algorithmic complexity of collections?**

The algorithmic complexity of a collection refers to the time and space efficiency of the operations performed on the collection. It is important to consider the algorithmic complexity when choosing a collection class because it can affect the performance of your application.

The algorithmic complexity of different collections varies depending on the type of operation performed. For example, the time complexity of adding an element to an ArrayList is O(1) on average, while the time complexity of adding an element to a LinkedList is O(n) on average.

The following table shows the algorithmic complexity of some common mutable collection types in C#:



**TASK 7 – Unit Testing**

* **What is a unit test? What other types of tests exist?**A **unit test** is a type of software testing where individual units or components of a software are tested to validate that each unit of the software code performs as expected. Unit tests isolate a section of code and verify its correctness. A unit may be an individual function, method, procedure, module, or object.

**Functional Testing**: This type of testing verifies that the software application functions as expected and meets the requirements specified by the client. It is performed by testers and not developers.

**Integration Testing**: This type of testing verifies that different modules or components of the software application work together as expected. It is performed after unit testing and before system testing.

**End-to-End Tests**: This type of testing verifies that the entire software application works as expected and meets the requirements specified by the client. It is performed after integration testing.

**Acceptance Testing**: This type of testing verifies that the software application meets the requirements specified by the client and is ready for deployment. It is performed by the client or end-users.

* **What is testing pyramid?**

The **testing pyramid** is a metaphorical model that helps software development teams to group software tests into different buckets of granularity. It also provides an idea of how many tests should be in each of these groups.

When you go up on the testing pyramid, the tests become more complex and take more time to run. However, they also provide more coverage and find more bugs. When you go down on the testing pyramid, the tests become simpler and faster to run. They provide less coverage but are easier to maintain and change.

* **What is TDD and BDD?  
  Test Driven Development (TDD)** and **Behavior Driven Development (BDD)** are two software development techniques.

**TDD** is a development technique where developers create test cases first and then write the code underlying those test cases. The teams that implement TDD take more time for development, but they tend to find very few defects. TDD results in improved quality of code and the code that is more reusable and flexible. TDD also helps in achieving high test coverage of about 90%.

**BDD** is a test-first approach that differs from TDD by testing the system’s actual behavior from the end-users perspective. BDD means building a shared understanding/agreement with a wider group of stakeholders using plain human-readable language to describe how the code/application needs to behave. BDD is meant to test how an application behaves from the perspective of the end-user.

* **What is a mock, stub and fake?**

In software testing; **mocks**, **stubs**, and **fakes** are all types of test doubles. Test doubles are objects that replace real dependencies in a test to isolate the code under test and make it easier to test.

A **mock** is a fake object that is used to verify interactions between the code under test and its dependencies. Mocks are used to ensure that the code under test is calling the correct methods on its dependencies with the correct arguments.

A **stub** is a fake object that provides canned answers to calls made during the test. Stubs are used to replace real dependencies with predictable behavior so that the code under test can be tested in isolation.

A **fake** is a fake object that has a working implementation but takes some shortcuts that make it unsuitable for production use. Fakes are used to replace real dependencies with simpler implementations so that the code under test can be tested in isolation.

* **Which test frameworks in C# do you know?**

There are several testing frameworks available for C#. Some of the popular ones are:

**MSTest**: This is a testing framework that comes with Microsoft Visual Studio. It is used for unit testing and provides annotations to mark test classes and methods.  
  
**NUnit**: This is a popular open-source testing framework for C#. It provides a rich history of unit testing along with C# functionality.  
  
**xUnit**: This is another open-source testing framework for C#. It is based on the .NET framework and provides a simple, easy-to-use interface.

* **What best practices of unit testing do you know? What is AAA, FIRST?  
  Test only one thing**: A unit test should test only one scenario or behavior. This makes it easier to identify the cause of a failure and fix it quickly.  
    
  **Use meaningful names**: Use descriptive names for your tests and test methods. This makes it easier to understand what the test is doing and what it is testing.  
    
  **Test all code paths**: Make sure that your tests cover all code paths in your application. This includes error handling, edge cases, and other scenarios that may not be immediately obvious.  
    
  **Use the AAA pattern**: is a pattern for structuring unit tests that helps make them more readable and maintainable. The pattern consists of three phases: Arrange, Act, and Assert. The Arrange phase sets up the system under test and any necessary dependencies. The Act phase performs the action you want to test. The Assert phase checks that the result of the action is correct.  
    
  **Use the FIRST principles**:is a set of principles for writing effective unit tests.  
  **Fast**: Unit tests should be fast so that they can be run frequently.  
  **Isolated**: Unit tests should be isolated from other tests so that they do not interfere with each other.  
  **Repeatable**: Unit tests should be repeatable so that they produce consistent results.  
  **Self-validating**: Unit tests should be self-validating so that they can be run automatically without human intervention.  
  **Timely**: Unit tests should be written in a timely manner so that they can catch bugs early in the development process.

**TASK 8 – SQL Fundamentals**

**What is the difference between primary and secondary keys? How many primary keys are available for the table? How many secondary keys are available for the table?**A **primary key** is a field or a combination of fields that uniquely identifies each record in a database table. It must contain unique values and cannot have NULL values. A table can have only one primary key constraint, which may consist of single or multiple fields. When multiple keys are used in a single primary key, these fields are called a composite key.

A **secondary key** is a field or a combination of fields that is not selected as the primary key but is capable of identifying each record uniquely. It is also known as an alternate key. A table can have more than one secondary key.

In terms of the number of primary keys available for a table, a table can have only one primary key constraint, which may consist of single or multiple fields.  
  
A table can have multiple **secondary keys**. Unlike primary keys, secondary keys are not unique and can have NULL values. They are used to speed up the querying of data and enforce data integrity constraints. **What is the difference between clustered and non-clustered indexes?**A **clustered index** is an index that sorts and stores the data rows in a table based on their key values. The key values are the columns included in the index definition. There can be only one clustered index per table, and the data rows themselves can be stored in only one order. When a table has a clustered index, it is called a clustered table. If a table has no clustered index, its data rows are stored in an unordered structure called a heap.

A **non-clustered index** is an index that has a structure separate from the data rows. A non-clustered index contains the non-clustered index key values and each key value entry has a pointer to the data row that contains the key value. The pointer from an index row in a non-clustered index to a data row is called a row locator. The structure of the row locator depends on whether the data pages are stored in a heap or a clustered table. For a heap, a row locator is a pointer to the row. For a clustered table, the row locator is the clustered index key.

In summary, both types of indexes are used to speed up querying of data and enforce data integrity constraints. However, they differ in how they store and sort data rows. **What is a transaction?**A **database transaction** is a logical unit of work that is independently executed for data retrieval or updates in a database management system. It is a sequence of one or more database operations that are treated as a single unit of work. Transactions are used to ensure data consistency and integrity in a database, even in the event of system failures.

In a database transaction, the following four properties are known as the **ACID** properties:

A simple example of a database transaction is transferring money from one bank account to another. The transaction consists of two operations: subtracting the amount to be transferred from one account and adding that same amount to the other. The entire transaction must either be completed successfully or not at all.  
 **What is a trigger? How to create a trigger?**A **database trigger** is a stored procedure that is automatically executed in response to certain events on a particular table or view in a database. The trigger is mostly used for maintaining the integrity of the information on the database.

For example, a human resources (HR) application requires that every employee manager receive an informational e-mail immediately after an employee’s leave request is submitted. When a record is written into a table that stores employee leave requests, a created trigger fires and invokes the e-mail-sending procedure to the manager. Another common trigger use is to save important original data, in its unchanged state, to maintain an audit trail or ensure that the original data remains accessible in the event of accidental changes.

To create a database trigger, you can use the **CREATE TRIGGER** statement. The syntax of the **CREATE TRIGGER** statement varies depending on the database management system you are using.  
  
Here’s an example of how to create a trigger in SQL Server:

**CREATE TRIGGER trg\_employee\_leave\_request**

**ON employee\_leave\_requests**

**AFTER INSERT**

**AS**

**BEGIN**

**-- Send an email to the employee's manager**

**END;**

This creates a trigger named **trg\_employee\_leave\_request** on the **employee\_leave\_requests** table that fires after an **INSERT** operation is performed on the table.  
 **What is the difference between INNER JOIN, RIGHT JOIN and LEFT JOIN?**In SQL, **joins** are used to combine data from two or more tables based on a related column between them. There are three types of joins: **INNER JOIN**, **LEFT JOIN**, and **RIGHT JOIN**.

An **INNER JOIN** returns only the rows that have matching values in both tables. In other words, it returns the intersection of the two tables.

A **LEFT JOIN** returns all the rows from the left table and the matching rows from the right table. If there is no match in the right table, NULL values are returned for those columns.

A **RIGHT JOIN** returns all the rows from the right table and the matching rows from the left table. If there is no match in the left table, NULL values are returned for those columns. **What is a stored procedure? How to create a SP?**A **stored procedure** is a named collection of SQL statements and procedural logic that is compiled, verified, and stored in the server database. It is a precompiled SQL code that can be saved and reused over and over again.To create a stored procedure, you can use the **CREATE PROCEDURE** statement. The syntax of the **CREATE PROCEDURE** statement varies depending on the database management system you are using.  
  
Here’s an example of how to create a stored procedure in SQL Server:

**CREATE PROCEDURE sp\_GetCustomers**

**AS**

**BEGIN**

**SELECT \* FROM Customers**

**END;**

This creates a stored procedure named **sp\_GetCustomers** that retrieves all records from the **Customers** table.  
 **What is a view? How to create a view?**A **view** is a virtual table that is based on the result of an SQL query. It contains rows and columns, just like a real table, but does not store any data itself. Instead, it retrieves data from one or more tables in the database and presents it as if it were a single table.

Views are used to simplify complex queries, provide an additional layer of security, and improve performance by reducing the amount of data that needs to be retrieved from the database.

To create a view, you can use the**CREATE VIEW** statement. The syntax of the **CREATE VIEW** statement varies depending on the database management system you are using.   
  
Here’s an example of how to create a view in SQL Server:

**CREATE VIEW view\_name AS**

**SELECT column1, column2, ...**

**FROM table\_name**

**WHERE condition;**

This creates a view named **view\_name** that retrieves data from **table\_name** based on the specified condition.

**What is the difference between view and table?**A **table** is a database object that stores data in rows and columns. It is a physical structure that contains the actual data of the database. Tables are used to store, organize, and manipulate data in a relational database management system (RDBMS).

On the other hand, a **view** is a virtual table that is based on the result of an SQL query. It does not store any data itself but retrieves data from one or more tables in the database and presents it as if it were a single table. Views are used to simplify complex queries, provide an additional layer of security, and improve performance by reducing the amount of data that needs to be retrieved from the database.

In summary, the main difference between a view and a table is that a table stores actual data, while a view does not. A view is simply a saved SQL query that retrieves data from one or more tables in the database.

**TASK 9 – Logging**

**What is logging?**In .NET, **logging** is the process of recording events that occur in a computer system, such as problems, errors, or just information on current operations. The logging system in .NET is designed to help monitor application behavior and diagnose issues. The **ILogger API** is used to write logs in .NET. The ILogger API is a high-performance, structured logging system that can be used to write logs to different destinations by configuring different logging providers. Basic logging providers are built-in, and there are many third-party providers available as well. The logs can be written to a file, a dedicated logging system, or a log management software, where it is stored in a database or on a different computer system. **What information needs to be logged?**  
The data that needs to be logged depends on the system being used and the purpose of the logs. In general, logs should capture data that can help diagnose problems, monitor system behavior, and provide an audit trail. Here are some examples of data that can be logged:

1. Date and time of access for every user
2. User IP address
3. Number of consecutive login attempts
4. Session length
5. Data entered in form fields (to detect SQL injection attacks)
6. HTTP headers (to track user agents, referrers, etc.)
7. Error messages and stack traces
8. Performance metrics (response time, CPU and memory usage, etc.)

It’s important to note that logging sensitive data such as passwords, credit card numbers, or other personally identifiable information (PII) should be avoided. The logs should be stored securely and access to them should be restricted to authorized personnel only.

**What log levels do you know and how do they differ from each other?  
  
TRACE**: The most verbose level. Used for development and seldom enabled in production. It’s intended for logging detailed information about the system for debugging purposes.

**DEBUG**: Debugging the application behavior from internal events of interest.

**INFO**: Information that highlights progress or application lifetime events.

**WARN**: Warnings about validation issues or temporary failures that can be recovered; may indicate future problems.

**ERROR**: Denotes a serious problem that has to be dealt with.

**FATAL**: The most severe level. Indicates that the system is unusable and requires immediate attention.  
  
**What is ILogger in .NET Core?**In .NET Core, **ILogger** is an interface that provides a way to write logs in a structured manner. It is part of the Microsoft.Extensions.Logging namespace and is used to log messages at different levels of severity, such as Information, Warning, Error, and Critical. ILogger is designed to be used with a logging framework that implements the interface, such as Serilog, NLog, or Log4Net.

The ILogger interface provides several methods for logging messages, including LogDebug, LogInformation, LogWarning, LogError, and LogCritical. Each method takes a message string and an optional set of parameters that can be used to format the message. The ILogger interface also provides a way to create scopes, which can be used to group related log messages together.

**Is it possible to stop logging with Serilog? If "yes", how can you do it?**Yes, it is possible to stop logging with Serilog. One way to do this is to set the minimum log level to a higher level than the level of the logs you want to suppress. For example, if you want to suppress all logs with a level of Debug or lower, you can set the minimum log level to Information.  
  
Another way is to override the minimum log level for different sources, so that you can continue to receive Information-level logs written by some sources, whilst ignoring Information-level logs written by the host. You can also use **Serilog.Filters.Expressions** plugin to log to different files based on the source context. **What information is good to include in a log message?  
Log level:** Indicates the severity of the logged event and is used to filter out less important log messages.

**Caller identity:** The identity (or principal) of the caller is also an important piece of information.

**Thread ID:** Enterprise applications are often executed in a multithreaded environment. With thread ID information, you can distinguish one request from another.

**Timestamp:** Generally, users can only approximate the time at which a problem occurred. Timestamps can help to pinpoint the exact time of an event.

**Source code information:** This includes class name, method name, and line number.  
 **How can Exceptions be logged?**To log an exception, you can use the **LogError** method of the **ILogger** object. The **LogError** method takes an exception object as a parameter and logs the exception at the Error level. **What is the default log level?**The default log level depends on the logging framework being used. In general, most logging frameworks have a default log level of **Warning** or **Error**. This means that only logs with a severity level of Warning or higher (or Error or higher) will be recorded by default. **Where can custom logging provider and corresponding logger be registered?**In .NET Core C#, custom logging providers and corresponding loggers can be registered in the **ConfigureLogging** method of the **HostBuilder** class. The **ConfigureLogging** method takes an **Action<ILoggingBuilder>** parameter that can be used to configure the logging system.

**TASK 10 – Serialization**

What is serialization?  
Serialization is the process of translating a data structure or object state into a format that can be stored or transmitted and reconstructed later. The resulting series of bits can be used to create a semantically identical clone of the original object. Serialization is useful for storing data in databases or on hard disk drives, transferring data over computer networks, and detecting changes in time-varying data. The opposite operation of serialization is deserialization, which extracts a data structure from a series of bytes.

Give examples where serialization is used?  
Serialization is used in many web applications, including:

**Web services**: Serialization is used to transfer data between web services and clients.

**Caching**: Serialization is used to cache data in web applications.

**Session management**: Serialization is used to store session data in web applications.

**Data transfer**: Serialization is used to transfer data between web applications and databases.

**State management**: Serialization is used to manage the state of web applications.

**Web APIs**: Serialization is used to serialize and deserialize data in web APIs.

When binary/XML/Json serialization need to be used? What are the main advantages of binary/XML/Json serialization?  
Binary serialization is typically used when speed and efficiency are important, such as when transferring large amounts of data or when the data needs to be transmitted over a high-bandwidth network connection. Binary serialization preserves type fidelity, which means that the complete state of the object is recorded and when you deserialize, an exact copy is created.

XML serialization provides more readable code, as well as greater flexibility of object sharing and usage for interoperability purposes. XML is an open standard, which makes it an attractive choice for sharing data across the Web.

JSON serialization maps .NET objects to and from JavaScript Object Notation (JSON). JSON is an open standard that’s commonly used to share data across the web. The JSON serializer serializes public properties by default, and can be configured to serialize private and internal members as well. JSON is a lightweight format, which makes it ideal for use in web applications.

In summary, binary serialization is ideal for high-performance applications, XML serialization is useful for interoperability purposes, and JSON serialization is lightweight and ideal for use in web applications.

When is custom serialization needed? How can custom serialization be implemented?

Custom serialization is needed when the default serialization provided by the programming language or framework does not meet the specific requirements of the application. For example, if you need to serialize an object that contains sensitive data, you may want to implement custom serialization to encrypt the data before it is stored or transmitted.

In .NET, custom serialization can be implemented by defining the **ISerializable** interface and implementing the **GetObjectData** and **ISerializable** constructor methods in the class that needs to be serialized. The **GetObjectData** method is called by the serialization process to write the object’s state to a stream, and the **ISerializable** constructor is called by the deserialization process to read the object’s state from a stream.

**TASK 11 – OOP Fundamentals and Design Principles**

What are the four principles of object-oriented programming? Can you explain each in your own words?

**Encapsulation:** This principle is all about hiding the complexity of an object's internal workings from the outside world. It means that an object should have a well-defined interface that allows other objects to interact with it, but the details of how it works should be hidden.

**Inheritance:** This principle allows objects to inherit properties and methods from other objects. It's a way of reusing code and creating a hierarchy of objects that share common characteristics.

**Polymorphism:** This principle allows objects to take on different forms or behaviors depending on the context in which they are used. It's a way of making code more flexible and adaptable to different situations.

**Abstraction:** This principle is about focusing on the essential features of an object and ignoring the details that are not relevant to its use. It's a way of simplifying complex systems and making them easier to understand and work with.

By following these principles, object-oriented programming allows developers to create modular, reusable, and maintainable code that can be easily adapted to changing requirements.

What is multiple inheritance? How are we able to achieve it in .NET?

**Multiple inheritance** is a feature of object-oriented programming that allows a class to inherit properties and methods from more than one parent class. This means that a single class can have multiple base classes, each of which contributes to its functionality.

**In .NET, multiple inheritance is not directly supported**, but it can be achieved through the use of interfaces. By implementing multiple interfaces, a class can effectively inherit functionality from multiple sources.

For example, suppose we have two classes, A and B, each with their own set of properties and methods. We can create an interface I, that specifies the common functionality between the two classes. Then, we can have a third class, C, that implements both A and B as well as the interface I. This allows C to inherit functionality from both A and B, while still adhering to the contract specified by the interface.

By using interfaces in this way, we can achieve the benefits of multiple inheritance while avoiding some of the potential pitfalls, such as the diamond problem, which can arise when two parent classes have conflicting methods or properties.

What do S.O.L.I.D. principles stand for? Can you decipher each letter?

The SOLID principles are a set of five design principles that are intended to help developers create more maintainable and scalable software. The letters in SOLID stand for:

**S - Single Responsibility Principle:** This principle states that a class should have only one reason to change. In other words, a class should have only one responsibility or job, and it should be focused on that job.

**O - Open/Closed Principle:** This principle states that a class should be open for extension but closed for modification. In other words, we should be able to add new functionality to a class without changing its existing code.

**L - Liskov Substitution Principle:** This principle states that objects of a superclass should be able to be replaced with objects of a subclass without affecting the correctness of the program. In other words, subclasses should be able to be used in place of their parent classes without causing any problems.

**I - Interface Segregation Principle:** This principle states that a class should not be forced to implement interfaces that it does not use. In other words, we should break down interfaces into smaller, more specific ones so that classes can implement only the functionality they need.

**D - Dependency Inversion Principle:** This principle states that high-level modules should not depend on low-level modules, but both should depend on abstractions. In other words, we should design our code so that it is easy to change and maintain by depending on abstractions rather than concrete implementations.

By following these principles, developers can create software that is more modular, flexible, and maintainable, which can lead to fewer bugs, faster development, and easier maintenance over time.

What other principles do you know, do they conflict or complement each other?

There are many other software principles (like GRASP) and design patterns that developers can use to create high-quality software. Some of the most common ones include:

* **DRY (Don't Repeat Yourself):** This principle states that code should not be duplicated, but instead should be abstracted into reusable functions or modules. This principle complements the Single Responsibility Principle of SOLID, as both are focused on reducing code duplication and increasing modularity.
* **KISS (Keep It Simple, Stupid):** This principle states that code should be kept as simple as possible, and unnecessary complexity should be avoided. This principle complements the Open/Closed Principle of SOLID, as both are focused on creating code that is easy to understand and modify.
* **YAGNI (You Ain't Gonna Need It):** This principle states that code should not be written until it is actually needed. This principle complements the Interface Segregation Principle of SOLID, as both are focused on creating code that is lean and focused on the specific needs of the application.
* **Composition over Inheritance:** This principle states that code should favor composition (combining smaller, simpler objects) over inheritance (creating complex hierarchies of objects). This principle complements the Liskov Substitution Principle of SOLID, as both are focused on creating code that is flexible and adaptable to changing requirements.

While these principles and patterns are generally complementary, there can be some conflicts or overlaps between them depending on the specific situation. For example:

1. **The Single Responsibility Principle (SRP) and the Open/Closed Principle (OCP)** can sometimes conflict with each other. If a class has only one responsibility, it may be difficult to extend it without modifying its existing code, which violates the OCP. On the other hand, if a class is designed to be open for extension, it may end up with multiple responsibilities, which violates the SRP.
2. **The Dependency Inversion Principle (DIP) and the Composition over Inheritance principle** can sometimes overlap with each other. Both principles are focused on creating code that is flexible and adaptable to changing requirements. However, DIP emphasizes the use of abstractions and interfaces to achieve this, while Composition over Inheritance emphasizes the use of smaller, simpler objects that can be combined in different ways.
3. **The Keep It Simple, Stupid (KISS) principle and the You Ain't Gonna Need It (YAGNI) principle** can sometimes conflict with each other. KISS emphasizes the importance of keeping code simple and easy to understand, while YAGNI emphasizes the importance of not writing code until it is actually needed. In some cases, writing code that is simple and easy to understand may require adding functionality that is not strictly necessary, which violates the YAGNI principle.

Despite these potential conflicts or overlaps, these principles and patterns are all focused on creating high-quality software that is maintainable, scalable, and easy to understand. By using a combination of these principles and patterns, developers can create software that is both efficient and effective.

**TASK 12 – LINQ**

What is LINQ? What is it used for?  
LINQ stands for Language Integrated Query. It is a feature in C# programming language that allows you to query data from different data sources such as databases, XML documents, and collections using a unified syntax. LINQ provides a set of standard query operators that can be used to perform various operations on data such as filtering, sorting, grouping, and aggregating.

LINQ is used to simplify the process of querying data from different data sources. It allows developers to write queries using a single syntax, regardless of the data source. This makes it easier to write and maintain code, as well as improving the performance of the application. LINQ also provides type safety, which means that errors can be caught at compile time rather than at runtime.

When are query expressions executed? What is materialization in LINQ? Name query methods which lead to query execution.  
Query expressions in LINQ are executed when they are enumerated. This means that the query is not executed until the results are actually needed, such as when the results are iterated over in a foreach loop or when the ToList or ToArray method is called. This is known as deferred execution.

Materialization in LINQ refers to the process of converting the results of a query into a concrete data type, such as a list or an array. This is necessary because the results of a query are initially represented as an IEnumerable or IQueryable, which are lazy evaluated and do not contain the actual data until they are enumerated.

Some of the query methods that lead to query execution and materialization include:

1. ToList() - Converts the results of a query to a List<T>.
2. ToArray() - Converts the results of a query to an array.
3. Count() - Returns the number of elements in the sequence.
4. First() - Returns the first element of the sequence.
5. Single() - Returns the only element of the sequence, or throws an exception if there is not exactly one element.
6. Any() - Returns true if the sequence contains any elements, otherwise false.
7. Sum() - Returns the sum of the numeric values in the sequence.
8. Average() - Returns the average of the numeric values in the sequence.
9. Max() - Returns the maximum value in the sequence.
10. Min() - Returns the minimum value in the sequence.

It's important to note that not all LINQ methods lead to query execution and materialization. Some methods, such as Where() and Select(), only modify the query and do not execute it.

What is query syntax? What is the method syntax?  
Query syntax and method syntax are two ways to write LINQ queries in C#.

Query syntax is a declarative syntax that uses a SQL-like syntax to write queries. It uses keywords such as from, where, select, and orderby to construct the query. Here's an example of a LINQ query written in query syntax:

  var result = from student in students

where student.Age > 18

orderby student.LastName

select student;

This query selects all students whose age is greater than 18, orders the results by last name, and returns the selected students.

Method syntax, on the other hand, is an imperative syntax that uses extension methods to write queries. It uses methods such as Where(), Select(), and OrderBy() to construct the query. Here's an example of the same LINQ query written in method syntax:

var result = students.Where(s => s.Age > 18)

.OrderBy(s => s.LastName)

.Select(s => s);  
  
This query does the same thing as the previous query, but it uses method chaining to construct the query.

Both query syntax and method syntax are equivalent and can be used interchangeably. The choice of syntax is a matter of personal preference and coding style.

When is query syntax converted to method calls?  
Query syntax is converted to method calls at compile time. When you write a LINQ query using query syntax, the C# compiler translates it into method calls using the extension methods provided by the LINQ library. This process is called query comprehension. 

If query expression syntax does not have some operations, for example Count or Max, how to combine query syntax with method calls?  
To do this, you can use the let keyword in the query expression syntax to define a new variable that represents the result of a method call. You can then use this variable in the rest of the query expression.

Which are basic LINQ query operations?  
There are several basic LINQ query operations that can be used to manipulate data in C#. These operations are provided by the standard query operators in the LINQ library and include:

1. Filtering: This operation is used to select a subset of elements from a collection based on a specified condition. The Where() method is used for filtering.
2. Projection: This operation is used to transform the elements of a collection into a new form. The Select() method is used for projection.
3. Ordering: This operation is used to sort the elements of a collection based on a specified criteria. The OrderBy() and OrderByDescending() methods are used for ordering.
4. Grouping: This operation is used to group the elements of a collection based on a specified key. The GroupBy() method is used for grouping.
5. Joining: This operation is used to combine two collections based on a common key. The Join() method is used for joining.
6. Aggregation: This operation is used to perform a calculation on a collection, such as calculating the sum, average, or maximum value of a set of numbers. The Aggregate() method is used for aggregation.
7. Set operations: These operations are used to perform set operations on collections, such as union, intersection, and difference. The Union(), Intersect(), and Except() methods are used for set operations.

How do you transform data with LINQ?  
You can transform data with LINQ using the Select() method, which is used for projection. The Select() method is used to transform the elements of a collection into a new form by applying a transformation function to each element.

The transformation function is a lambda expression that takes an element of the collection as input and returns a new object that represents the transformed element. The new object can be of any type, including an anonymous type.

What is LINQ to Objects?  
LINQ to Objects is a LINQ provider that allows you to query data from in-memory collections such as arrays, lists, and dictionaries using LINQ syntax. It is the most commonly used LINQ provider and is included in the .NET Framework.

With LINQ to Objects, you can use the standard query operators provided by the LINQ library to perform various operations on in-memory collections, such as filtering, sorting, grouping, and aggregating. These operations are performed using deferred execution, which means that the query is not executed until the results are actually needed. 

Can you project a query to an anonymous type?  
Yes, you can project a LINQ query to an anonymous type using the Select() method. An anonymous type is a type that is defined at compile time and has no name. It is used to create objects that have a specific set of properties, without having to define a new class.

**TASK 13 – ADO.NET Fundamentals**

What is ADO.NET?

ADO.NET is a data access technology that is used to communicate with databases from .NET applications. It provides a set of classes that allow you to connect to a database, execute queries, and retrieve data. ADO.NET is a part of the .NET Framework and is used by developers to create data-driven applications. It supports a variety of databases including SQL Server, Oracle, MySQL, and more.

Explain the difference between terms: connected model and disconnected model?

Connected model refers to a way of working with a database where a connection to the database is established and maintained throughout the lifetime of the application. In this model, the application sends a query to the database, retrieves the results, and then closes the connection. This means that the application is "connected" to the database for the entire duration of the operation. This model is useful when you need to perform a series of operations on the database that are related to each other and need to be executed in a specific order.

On the other hand, disconnected model refers to a way of working with a database where a connection to the database is established only when data needs to be retrieved or updated. In this model, the application connects to the database, retrieves the data, and then disconnects from the database. The application then works with the data in memory, and when it's time to update the database, it connects again, updates the data, and then disconnects again. This model is useful when you need to work with data offline or when you need to work with a large amount of data that would be impractical to keep in memory for the entire duration of the application.

In summary, the main difference between connected and disconnected models is that in the connected model, the application maintains a connection to the database for the entire duration of the operation, while in the disconnected model, the application connects to the database only when it needs to retrieve or update data.

When do we need to use a DataReader and a DataSet?

Both DataReader and DataSet are used to retrieve data from a database in ADO.NET, but they have different use cases.

A DataReader is a forward-only, read-only stream of data from a database. It is used when you need to quickly retrieve a large amount of data and read it sequentially. DataReader is a lightweight object that is optimized for performance and memory usage. It is ideal for scenarios where you need to read data once and do not need to update it. DataReader is also useful when you need to process data in a streaming fashion, such as when you are exporting data to a file or sending it over a network.

A DataSet, on the other hand, is an in-memory representation of data that can be used to store and manipulate data from multiple tables. It is used when you need to work with data in a disconnected manner, such as when you need to cache data for later use or when you need to work with data offline. A DataSet can contain multiple DataTables, which can be related to each other using DataRelations. A DataSet can also be used to update data back to the database using a DataAdapter.

In summary, you would use a DataReader when you need to quickly retrieve a large amount of data and read it sequentially, and you would use a DataSet when you need to work with data in a disconnected manner, such as when you need to cache data for later use or when you need to work with data offline.

What do we need to use data providers for?

Data providers are used in ADO.NET to provide a consistent way of accessing data from different types of data sources, such as databases, XML files, and web services. A data provider is a set of classes that implement the common ADO.NET interfaces, such as IDbConnection, IDbCommand, and IDbDataAdapter, and provide specific implementations for a particular data source.

Data providers are needed because different data sources have different ways of storing and retrieving data, and they may have different performance characteristics and security requirements. For example, a SQL Server database may use T-SQL for querying data, while an Oracle database may use PL/SQL. A data provider for SQL Server would provide an implementation of IDbConnection, IDbCommand, and IDbDataAdapter that is optimized for SQL Server, while a data provider for Oracle would provide an implementation that is optimized for Oracle.

By using data providers, developers can write code that is independent of the underlying data source. This means that the same code can be used to access data from different types of data sources, without having to change the code. Data providers also provide a consistent way of working with data, which makes it easier for developers to learn and use ADO.NET.

In summary, data providers are used in ADO.NET to provide a consistent way of accessing data from different types of data sources, and to abstract away the differences between different data sources.

What do we need to use transactions for?

Transactions are used in ADO.NET to ensure that a series of database operations are executed as a single, atomic unit of work. A transaction is a sequence of one or more database operations that are executed as a single, indivisible unit. Transactions are used to ensure that all the operations in the transaction are either completed successfully or rolled back if any of the operations fail.