**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#:

