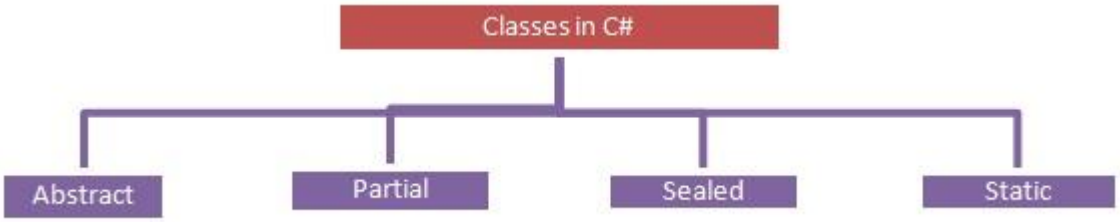
2. Advance C#

2.1 Types class

- Types of classes in c# :



- Abstract Class :
    - An abstract class is a class that provides a common definition to the subclasses, and this is the type of class whose object is not created.
    - We cannot create an object of an abstract class.
    - It must be inherited in a subclass if you want to use it.
    - An Abstract class contains both abstract and non-abstract methods.
    - The methods inside the abstract class can either have an or no implementation.
    - We can inherit two abstract classes; in this case, implementation of the base class method is optional.
    - An Abstract class has only one subclass.
    - Methods inside the abstract class cannot be private.
    - If there is at least one method abstract in a class, then the class must be abstract.
    - For example

```
abstract class Accounts
{
}
```

- Partial Class :
    - It is a type of class that allows dividing their properties, methods, and events into multiple source files, and at compile time, these files are combined into a single class.
    - All the parts of the partial class must be prefixed with the partial keyword.
    - If you seal a specific part of a partial class, the entire class is sealed, the same as for an abstract class.
    - Inheritance cannot be applied to partial classes.
    - The classes written in two class files are combined at run time.

- For example

```
partial class Accounts
{
}
```

- Sealed Class :
  - A Sealed class is a class that cannot be inherited and used to restrict the properties.
  - A Sealed class is created using the sealed keyword.
  - Access modifiers are not applied to a sealed class.
  - To access the sealed members, we must create an object of the class.
  - For example

```
sealed class Accounts
{
}
```

- Static Class :
  - It is the type of class that cannot be instantiated. In other words, we cannot create an object of that class using the new keyword, such that class members can be called directly using their name.
  - It was created using the static keyword.
  - Only static members are allowed; in other words, everything inside the class must be static.
  - We cannot create an object of the static class.
  - A Static class cannot be inherited.
  - It allows only a static constructor to be declared.
  - The static class methods can be called using the class name without creating the instance.
  - For example

```
static class Accounts
{
}
```

2.2 Generics

- Generics in C# provide a way to create classes, interfaces, and methods with placeholders for the data types they work with. This allows you to write code that can work with any data type, providing flexibility and type safety.

- Generics are extensively used in collections (such as List, Dictionary, etc.) and other scenarios where a common functionality is needed for different types.

Generic Classes:

- Syntax :

```
public class MyClass<T>
{
    private T value;
    public MyClass(T val)
    {
        value = val;
    }
    public T GetValue()
    {
        return value;
    }
}
```

Generic Methods :

- You can create generic methods inside non-generic classes as follows:

```
public class MyUtility
{
    public T Add<T>(T a, T b)
    {
        dynamic dynamicA = a;
        dynamic dynamicB = b;
        return dynamicA + dynamicB;
    }
}
```

Generic Interfaces :

- Syntax :

```
public interface IRepository<T>
{
    void Add(T item);
    T GetById(int id);
}
```

Constraints :

- You can use constraints to specify requirements on the generic type:

  public class MyClass<T> where T : IComparable { // Code here }

- This example ensures that T must implement the IComparable interface.

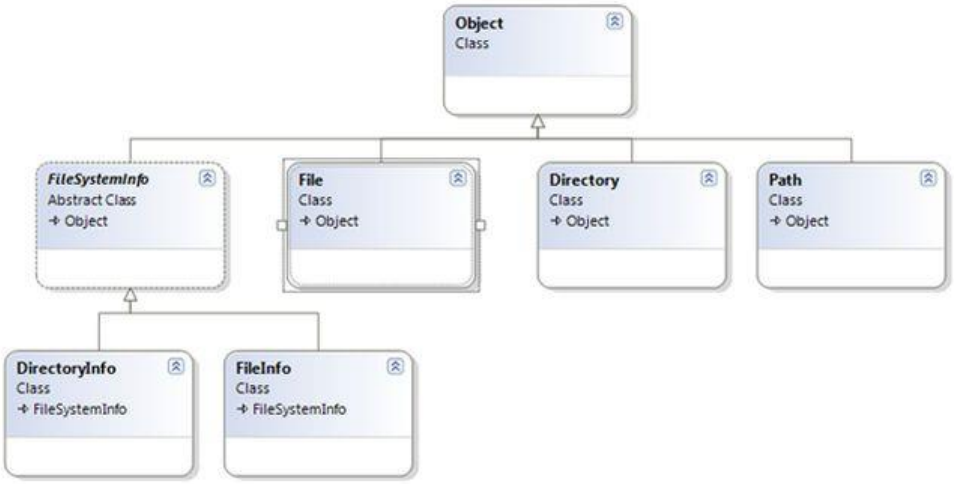
Covariance and Contravariance :

- Generics in C# support covariance and contravariance, allowing more flexibility when working with generic types. This is achieved using the out and in keywords.
- Syntax :

```
// Covariant interface
public interface IMyCovariant<out T>
 {
        T GetItem();
}
// Contravariant interface
public interface IMyContravariant<in T>
{
        void SetItem(T item);
}
```

2.3 File system in Depth

- The System.IO namespace provides four classes that allow you to manipulate individual files, as well as interact with a machine directory structure. The Directory and File directly extends System.Object and supports the creation, copying, moving and deletion of files using various static methods. They only contain static methods and are never instantiated. The FileInfo and DirecotryInfo types are derived from the abstract class FileSystemInfo type and they are typically, employed for obtaining the full details of a file or directory because their members tend to return strongly typed objects. They implement roughly the same public methods as a Directory and a File but they are stateful and the members of these classes are not static.



- The following table outlines the core members of this namespace,

Class Types	Description
Directory/ DirectoryInfo	These classes support the manipulation of the system directory structure.
DriveInfo	This class provides detailed information regarding the drives that a given machine has.
FileStream	This gets you random file access with data represented as a stream of bytes.
File/FileInfo	These sets of classes manipulate a computer's files.
Path	It performs operations on System.String types that contain file or directory path information in a platform-neutral manner.
BinaryReader/ BinaryWriter	These classes allow you to store and retrieve primitive data types as binary values.
StreamReader/StreamWriter	Used to store textual information to a file.
StringReader/StringWriter	These classes also work with textual information. However, the underlying storage is a string buffer rather than a physical file.
BufferedStream	This class provides temp storage for a stream of bytes that you can commit to storage at a later time.

- The System.IO provides a class DriveInfo to manipulate the system drive related tasks. The DriveInfo class provides numerous details such as the total number of drives, calculation of total hard disk space, available space, drive name, ready status, types and so on.

- The following code snippets perform the rest of the DriveInfo class method operations
- The .NET framework provides the two rudimentary classes, DirectoryInfo and Directory, to do directory-related operations such as creation and deletion.

  DirectoryInfo Class :

  - The DirectoryInfo class contains a set of members for the creation, deletion, moving and enumeration over directories and subdirectories. Here, in the following code sample, display the information related to temp directory.

```
DirectoryInfo di=new DirectoryInfo(@"D:\temp");
Console.WriteLine("*******Direcotry Informations*******\n\n");
Console.WriteLine("Full Name={0}",di.FullName);
Console.WriteLine("Root={0}",di.Root);
Console.WriteLine("Attributes={0}", di.Attributes);
Console.WriteLine("Creation Time={0}", di.CreationTime);
Console.WriteLine("Name={0}", di.Name);
Console.WriteLine("Parent={0}", di.Parent);
```

  - Output :

  Directory Class :

  - The Directory class provides nearly the same functionality as DirecotryInfo. The Directory class typically returns string data rather than strongly typed DirectoryInfo objects. The following sample deletes the directory and subdirectory in the D drive.

```
static void Main(string[] args)
{
    DirectoryInfo di = new DirectoryInfo(@"d:\abc");
    Console.WriteLine("Name:{0}",di.FullName);
    Console.Write("Are you sure to Delete:");
    string str=Console.ReadLine();
    if (str == "y")
    {
```

```
            Directory.Delete(@"d:\abc", true);
        }
        Console.Write("Deleted.....");
    }
```

Reading and Writing to Files

- Reading and writing operations are done using a File object. The following code snippet reads a text file located in the machine somewhere.

```
private void button1_Click(object sender, EventArgs e)
{
   try
   {
      textBox2.Text = File.ReadAllText(txtPath.Text);
   }
   catch (FileNotFoundException)
   {
      MessageBox.Show("File not Found....");
   }
}
```

- Besides reading a file, we can write some contents over an existing text file by the File class WriteAllTest() method as in the following:
- File.WriteAllText(@"d:\test.txt", textBox2.Text);
- It takes a path to save the file and content input method medium such as a text box or any other control.

Stream

- The .NET provides many objects such as FileStream, StreamReader/Writer, BinaryReader/Writer to read from and write data to a file. A stream basically represents a chunk of data flowing between a source and a destination. Stream provides a common way to interact with a sequence of bytes regardless of what kind of devices store or display the bytes. The following table provides common stream member functions:

Methods	Description
Read()/ ReadByte()	Read a sequence of bytes from the current stream.
Write()/WriteByte()	Write a sequence of bytes to the current stream.
Seek()	Sets the position in the current stream.
Position()	Determine the current position in the current stream.
Length()	Return the length of the stream in bytes.

Flush()	Updates the underlying data source with the current state of the buffer and then clears the buffer.
Close()	Closes the current stream and releases any associated stream resources.

FileStream

- A FileStream instance is used to read or write data to or from a file. In order to construct a FileStream, first we need a file that we want to access. Second, the mode that indicates how we want to open the file. Third, the access that indicates how we want to access a file. And finally, the share access that specifies whether you want exclusive access to the file.

Enumeration	Values
FileMode	Create, Append, Open, CreateNew, Truncate, OpenOrCreate
FileAccess	Read, Write, ReadWrite
FileShare	Inheritable, Read, None, Write, ReadWrite

- The FileStream can read or write only a single byte or an array of bytes. You will be required to encode the System.String type into a corresponding byte array. The System.Text namespace defines a type named encoding that provides members that encode and decode strings to an array of bytes. Once encoded, the byte array is persisted to a file with the FileStream.Write() method. To read the bytes back into memory, you must reset the internal position of the stream and call the ReadByte() method. Finally, you display the raw byte array and the decoded string to the console.

```
using(FileStream fs=new FileStream(@"d:\ajay123.doc",FileMode.
Create))
{
  string msg = "first program";
  byte[] byteArray = Encoding.Default.GetBytes(msg);
  fs.Write(byteArray, 0, byteArray.Length);
  fs.Position = 0;
  byte[] rFile = new byte[byteArray.Length];
  for (int i = 0; i < byteArray.Length; i++)
```

```
    {
        rFile[i] = (byte)fs.ReadByte();
        Console.WriteLine(rFile[i]);
    }
Console.WriteLine(Encoding.Default.GetString(rFile));
}
```

2.4 Data Serialization

- Serialization is the process of converting the state of an object into a form that can be persisted or transported. The complement of serialization is deserialization, which converts a stream into an object. Together, these processes allow data to be stored and transferred.
- .NET features the following serialization technologies:
    - 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.
    - XML serialization serializes only public properties and fields and does not preserve type fidelity. This is useful when you want to provide or consume data without restricting the application that uses the data. Because XML is an open standard, it is an attractive choice for sharing data across the Web.
    - 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. This type of serialization is useful for preserving the state of an object between different invocations of an application. For example, you can share an object between different applications by serializing it to the Clipboard. You can serialize an object to a stream, to a disk, to memory, over the network, and so forth. Remoting uses serialization to pass objects "by value" from one computer or application domain to another.

2.5 Base library features

- The .NET framework provides a set of base class libraries which provide functions and features which can be used with any programming language which implements .NET, such as Visual Basic, C# (or course), Visual C++, etc.
- The base class library contains standard programming features such as Collections, XML, DataType definitions, IO (for reading and writing to files), Reflection and Globalization to name a few. All of which are contained in the System namespace. As well, it contain some non-standard features such as LINQ, ADO.NET (for database interactions), drawing capabilities, forms and web support.
- The below table provides a list each class of the base class library and a brief description of what they provide.

Base Class Library Namespace	Brief Description
System	Contains the fundamentals for programming such as the data types, console, match and arrays, etc.
System.CodeDom	Supports the creation of code at runtime and the ability to run it.
System.Collections	Contains Lists, stacks, hashtables and dictionaries
System.ComponentModel	Provides licensing, controls and type conversion capabilities
System.Configuration	Used for reading and writing program configuration data
System.Data	Is the namespace for ADO.NET
System.Deployment	Upgrading capabilities via ClickOnce
System.Diagnostics	Provides tracing, logging, performance counters, etc. functionality
System.DirectoryServices	Is the namespace used to access the Active Directory
System.Drawing	Contains the GDI+ functionality for graphics support
System.EnterpriseServices	Used when working with COM+ from .NET
System.Globalization	Supports the localization of custom programs

System.IO	Provides connection to file system and the reading and writing to data streams such as files
System.Linq	Interface to LINQ providers and the execution of LINQ queries
System.Linq.Expressions	Namespace which contains delegates and lambda expressions
System.Management	Provides access to system information such as CPU utilization, storage space, etc.
System.Media	Contains methods to play sounds
System.Messaging	Used when message queues are required within an application, superseded by WCF
System.Net	Provides access to network protocols such as SSL, HTTP, SMTP and FTP
System.Reflection	Ability to read, create and invoke class information.
System.Resources	Used when localizing a program in relation to language support on web or form controls
System.Runtime	Contains functionality which allows the management of runtime behavior.
System.Security	Provides hashing and the ability to create custom security systems using policies and permissions.
System.ServiceProcess	Used when a windows service is required
System.Text	Provides the StringBuilder class, plus regular expression capabilities
System.Threading	Contains methods to manage the creation, synchronization and pooling of program threads
System.Timers	Provides the ability to raise events or take an action within a given timer period.
System.Transactions	Contains methods for the management of transactions
System.Web	Namespace for ASP.NET capabilities such as Web Services and browser communication.
System.Windows.Forms	Namespace containing the interface into the Windows API for the creation of Windows Forms programs.

2.6 Lambda expression

- You use a lambda expression to create an anonymous function. Use the lambda declaration operator => to separate the lambda's parameter list from its body. A lambda expression can be of any of the following two forms:
- Expression lambda that has an expression as its body:
    - (input-parameters) => expression
- Statement lambda that has a statement block as its body:
    - (input-parameters) => { <sequence-of-statements> }
- To create a lambda expression, you specify input parameters (if any) on the left side of the lambda operator and an expression or a statement block on the other side.

2.7 LINQ

- Language-Integrated Query (LINQ) is the name for a set of technologies based on the integration of query capabilities directly into the C# language. Traditionally, queries against data are expressed as simple strings without type checking at compile time or IntelliSense support.
- When you write queries, the most visible "language-integrated" part of LINQ is the query expression. Query expressions are written in a declarative query syntax. By using query syntax, you perform filtering, ordering, and grouping operations on data sources with a minimum of code. You use the same query expression patterns to query and transform data from any type of data source.
- Query expressions query and transform data from any LINQ-enabled data source. For example, a single query can retrieve data from an SQL database and produce an XML stream as output.
- Query expressions use many familiar C# language constructs, which make them easy to read.
- The variables in a query expression are all strongly typed.
- A query isn't executed until you iterate over the query variable, for example in a foreach statement.
- At compile time, query expressions are converted to standard query operator method calls according to the rules defined in the C# specification. Any query that can be expressed by using query syntax can also be expressed by using method syntax. In some cases, query syntax is more readable and concise. In others, method syntax is more readable. There's no semantic or performance

difference between the two different forms. For more information, see C# language specification and Standard query operators overview.

- Some query operations, such as Count or Max, have no equivalent query expression clause and must therefore be expressed as a method call. Method syntax can be combined with query syntax in various ways.
- Query expressions can be compiled to expression trees or to delegates, depending on the type that the query is applied to. IEnumerable<T> queries are compiled to delegates. IQueryable and IQueryable<T> queries are compiled to expression trees. For more information, see Expression trees.
- Example :

```
int[] scores = [97, 92, 81, 60];
IEnumerable<int> scoreQuery =
   from score in scores
   where score > 80
   select score;
foreach (var i in scoreQuery)
{
   Console.Write(i + " ");
}
```

2.7 ORM Tool

- ORM is a technique that maps object-oriented models to relational database models. It enables programmers to use object-oriented constructs from a programming language to interact with databases. In other words, ORM is a way to use the database as an object-oriented data source.
- With ORM, developers can use programming languages to create and manipulate database objects instead of writing SQL queries. ORM tools provide a set of APIs that developers can use to interact with the database. These APIs abstract the complexity of working with databases, such as opening database connections, executing SQL statements, and managing transactions.
- ORM tools use metadata to map the database tables to objects in the programming language. Metadata contains information about the database schema, such as table names, column names, data types, primary and foreign keys, and relationships between tables.

- ORMlite, or Object-Relational Mapping Lite, is a lightweight ORM framework for .NET that simplifies database operations by allowing you to interact with databases using C# objects.
- Steps to implement ORMLite :

1. Install NuGet Package: ServiceStack.OrmLite
2. Set Up Database Connection: Configure the database connection in your Web.config file. You can define the connection string for your database.

```
<connectionStrings> <add name="MyDb"
connectionString="Server=your_server;Database=your_database;User
Id=your_username;Password=your_password;"
providerName="System.Data.SqlClient" /> </connectionStrings>
```

3. Define Model Classes: Create C# classes that represent your database tables. These classes should have properties that match the columns in your tables.
4. Initialize ORMLite: In your Global.asax.cs file or a similar location, initialize ORMLite and configure it to use your database connection.

```
public class WebApiApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        OrmLiteConfig.DialectProvider=SqlServerDialect.Provider;
        string connectionString =
        ConfigurationManager.ConnectionStrings["MyDb"].Connect
        ionString;
        OrmLiteConnectionFactory dbFactory = new
        OrmLiteConnectionFactory(connectionString,
        SqlServerDialect.Provider);
        OrmLiteConfig.TidyProviderFilters();
    }
}
```

5. Perform Database Operations in Web API Controllers: In your Web API controllers, you can now use ORMLite to perform database operations.

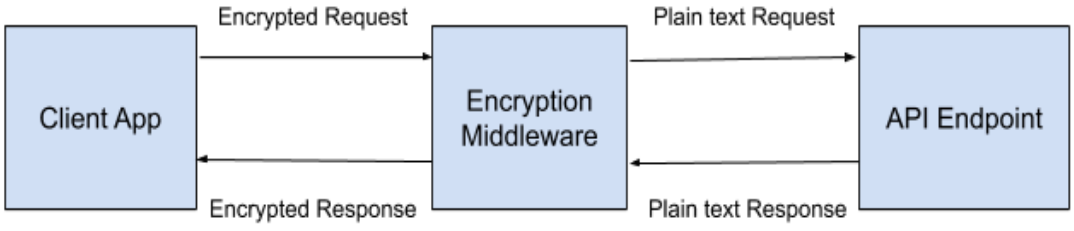
```
public class CustomerController : ApiController
{
    private readonly IDbConnectionFactory dbFactory;
    public CustomerController()
    {
```

```
                dbFactory = new OrmLiteConnectionFactory
                (ConfigurationManager.ConnectionStrings["MyDb"].
                ConnectionString, SqlServerDialect.Provider);
        }
        public IHttpActionResult GetCustomers()
        {
                using (var db = dbFactory.OpenDbConnection())
                {
                        List<Customer> customers=db.Select<Customer>();
                        return Ok(customers);
                }
    }
```

- You can extend the controller to include other CRUD operations such as insert, update, and delete.

2.8 Security and cryptography

- Nowadays, many applications are transferring sensitive data in a plain text format to and from other applications, which leads to data breach or leaks. We could prevent such data leaks by encrypting all the data before transfer it over the network.
- All the data transfers with the Client App will be encrypted. On the server-side, we would have an interceptor that would decrypt the incoming Request and encrypt the outgoing Response.



- To intercept the incoming and outgoing data, we need to create custom middleware, “EncryptionMiddleware” to encrypt the outgoing Response data and decrypt the incoming Request data.
- We have the decryption and encryption layer on the incoming Request body and outgoing Response body using DecryptStream and EncryptStream respectively. Also, we have DecryptString to decrypt the query string. We could customize the request pipeline to only execute data encryption on selective API routes.

2.9 Dynamic Type

- The dynamic type is a static type, but an object of type dynamic bypasses static type checking. In most cases, it functions like it has type object.
- The compiler assumes a dynamic element supports any operation. Therefore, you don't have to determine whether the object gets its value from a COM API, from a dynamic language such as IronPython, from the HTML Document Object Model (DOM), from reflection, or from somewhere else in the program.
- However, if the code isn't valid, errors surface at run time.
- Example :

```
dynamic d1 = 7;
d1 = "a string";
d1 = System.DateTime.Today;
```

2.10 Database with C# (CRUD)

- In C#, there are several libraries, classes, and namespaces that you can use to work with databases. Here are some of the key ones:
- ADO.NET (System.Data namespace):
    - SqlConnection: Represents a connection to a SQL Server database.
    - SqlCommand: Represents a SQL statement or stored procedure to execute against a SQL Server database.
    - SqlDataReader: Provides a way to read a forward-only stream of rows from a SQL Server database.
    - SqlDataAdapter: Represents a set of data commands and a database connection that are used to fill the DataSet and update a SQL Server database.
- Entity Framework (EF):
    - DbContext: Represents a session with the database, allowing you to query and save data.
    - DbSet<T>: Represents an entity set in the context, allowing you to query and save instances of the entity type.
    - Database: Provides methods for interacting with the database directly.
    - EntityConnection: Represents a connection to the database that can be used with Entity Framework.
- LINQ to SQL (System.Data.Linq namespace):
    - DataContext: Represents the main entry point for the LINQ to SQL framework.

- Table<T>: Represents a table for a specific type in the database.
- DataLoadOptions: Specifies the related data to load when retrieving entities.
- System.Data.Common namespace:
- DbConnection: Represents a connection to a database.
- DbCommand: Represents a command to execute against a database.
- DbDataReader: Represents a data reader forward-only stream of rows from a data source.

- Dapper:
  - Dapper is a micro ORM (Object-Relational Mapper) that simplifies data access in .NET applications.
  - Provides extension methods for IDbConnection to perform CRUD operations.
  - Here's a simple example using ADO.NET to connect to a SQL Server database:

```
using System;
using System.Data.SqlClient;
class Program
{
    static void Main()
    {
            string connectionString = "connection_string ";
            using (SqlConnection connection = new SqlConnection
            (connectionString))
            {
                connection.Open(); // Perform database operations using
                SqlCommand, SqlDataReader, etc. connection.Close();
            }
     }
 }
```