**C# Advance**

**Types class (i.e. Abstract, sealed etc.)**

**1. Abstract Class**

* Cannot be instantiated directly.
* Can have abstract (without implementation) and non-abstract methods.
* Used as a base class for other classes.

**2. Sealed Class**

* Cannot be inherited.
* Used to prevent further modification via inheritance.

**3. Static Class**

* Cannot be instantiated or inherited.
* Only contains static members.
* Used for utility/helper methods.

**4. Partial Class**

* Splits class definition across multiple files.
* Useful for large projects and auto-generated code.

**Generics**

Generics allow you to create reusable code that works with different data types while maintaining type safety.

**Generic Classes**

* Used to define a class with a placeholder type (T).

**Generic Methods**

* Allows methods to work with any data type.

**Generic Interfaces**

* Used to create flexible and reusable interfaces.

**Generic Constraints**

* Restricts the types that can be used with generics.

|  |  |
| --- | --- |
| Constraint | Description |
| where T : class | T must be a reference type. |
| where T : struct | T must be a value type. |
| where T : new() | T must have a parameterless constructor. |
| where T : BaseClass | T must inherit from BaseClass. |

**File system in Depth**

* **DirectoryInfo**: Provides instance-based operations for directories, such as creation, deletion, moving, and retrieving metadata.
* **FileInfo**: Works with files, allowing creation, deletion, copying, moving, and retrieving file properties.
* **StreamReader**: Used for reading text from files efficiently, supporting line-by-line or full-file reading.
* **StreamWriter**: Allows writing text to files, either overwriting or appending content.
* **FileStream**: Provides lower-level control for reading and writing both binary and text files, supporting various access modes and buffering.

**Data Serialization (JSON, XML)**

**1. JSON Serialization**

JSON (JavaScript Object Notation) is a lightweight data-interchange format that is easy for humans to read and write, and easy for machines to parse and generate. It is widely used for web APIs and data storage.

**Key Class for JSON Serialization**

* **JsonConvert** (from the **Newtonsoft.Json** library)

**Serialization Example:**

Convert an object into a JSON string:

JsonConvert.SerializeObject(book1);

**Deserialization Example:**

Convert a JSON string back into an object:

JsonConvert.DeserializeObject<Book>(jsonBook1);

**2. XML Serialization**

XML (Extensible Markup Language) is a markup language that defines rules for encoding documents in a format that is both human-readable and machine-readable. XML is commonly used for configuration files, data interchange, and web services.

**Key Classes for XML Serialization**

* **XmlSerializer** (from **System.Xml.Serialization**)

XmlSerializer xs = new XmlSerializer(typeof(List<Book>));

**// Serialization**: Converts the List<Book> to an XML file (books.xml).

xs.Serialize(fs, books);

**//** **Deserialization**: Reads the XML file and converts it back into a List<Book>.

List<Book> deserializedXMLBooks = (List<Book>)xs.Deserialize(fs);

**Lambda Expression**

A **lambda expression** is a shorthand syntax for writing anonymous methods (delegates) in C#. It is commonly used for inline code that can be passed around like methods or used in LINQ queries.

syntax: (parameters) => expression

Ex:

List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };

var evenNumbers = numbers.Where(n => n % 2 == 0).ToList();

**Extension Methods**

**Extension methods** allow you to add new functionality to existing types (classes, interfaces, or structs) without modifying their source code. They enable you to "extend" a type in a natural way, as if the method were part of the original type.

**Syntax:**

An extension method is defined in a static class with the this keyword preceding the first parameter, which specifies the type being extended.

**Key Points:**

1. **Must be defined in a static class**.
2. **The first parameter** specifies the type being extended (e.g., this string).
3. **Extension methods** are called like regular methods on the extended type.

Ex:

public static void Print<T>(this List<T> items)

{

foreach (var item in items)

{

Console.WriteLine(item);

}

}

**LINQ (Language-Integrated Query)**

LINQ (Language-Integrated Query) allows you to query various data sources (like arrays, collections, DataTables, databases, XML, etc.) directly within C#. It integrates query capabilities directly into C# syntax, providing a more readable and declarative way to interact with data.

**Syntax of LINQ**

LINQ provides two primary query syntax styles:

* **Query Syntax** (similar to SQL)
* **Method Syntax** (using LINQ extension methods)

Both styles achieve the same result; method syntax is more commonly used in modern development.

**Select in LINQ (Select , SelectMany)**

This is used to extract specific fields from a collection. For example, if you have a list of books, you can extract only the titles instead of the whole book objects.

**Filtering in LINQ (Where , OfType)**

Filters elements based on a condition. For example, from a list of employees, you can extract only those whose salary is above a certain threshold.

**Sorting in LINQ (OrderBy , OrderBYDescending , ThenBy , ThenBYDescending , Reverse)**

Allows ordering data in ascending or descending order. You can sort by any field, such as sorting a list of students by their grades.

**Quantifier Operations in LINQ**

These operations check for conditions in a collection:

* **Any**: Checks if there is at least one matching element.
* **All**: Checks if all elements satisfy a condition.
* **Contains**: Checks if element present or not

**Set Operations in LINQ**

These are used to compare and combine collections:

* **Union**: Combines two collections and removes duplicates.
* **Intersect**: Finds common elements between two collections.
* **Except**: Finds elements in one collection that don’t exist in another.
* **Distinct**: Removes duplicate values

**Partitioning in LINQ**

Divides data into smaller subsets:

* **Take**: Retrieves the first N elements.
* **Skip**: Ignores the first N elements.
* **TakeWhile/SkipWhile**: Works based on conditions instead of fixed numbers.

**Join in LINQ**

Combines data from multiple sources based on common fields. This is useful when working with related data, such as joining a list of books with a list of authors based on an author ID.

**Element Operations in LINQ**

These retrieve specific elements from a collection:

* **First**: Gets the first element.
* **Last**: Gets the last element.
* **Single**: Retrieves a specific element, assuming only one match exists.
* **Default versions**: Return null or default values if no match is found.

**Security & Cryptography**

Cryptography is essential for data protection, secure communication, and authentication. In C#, the System.Security.Cryptography namespace provides various encryption and hashing algorithms.

**1. Symmetric Encryption (Same Key for Encryption & Decryption)**

Used for fast encryption where the same key is used for both encryption and decryption.

**AES (Advanced Encryption Standard)**

* Most widely used and highly secure encryption algorithm.
* Supports key sizes 128-bit, 192-bit, and 256-bit.
* Used in SSL/TLS, VPNs, and data encryption.

**DES (Data Encryption Standard)**

* Older encryption standard (56-bit key) and is now considered weak due to brute-force attacks.
* Replaced by AES but still used in legacy systems.

**Rijndael**

* Predecessor of AES, supports variable block sizes (AES uses a fixed 128-bit block size).
* AES is a specific implementation of Rijndael with strict parameters.

**2. Asymmetric Encryption (Different Keys for Encryption & Decryption)**

Used for secure communication where encryption uses a public key, and decryption uses a private key.

**RSA (Rivest-Shamir-Adleman)**

* Widely used in secure communications, including SSL/TLS, email encryption, and digital signatures.
* Uses key pairs (public & private).
* Slower than symmetric encryption but offers better security.

**Dynamic Type**

The dynamic type in C# allows you to define variables without specifying their exact type at compile time. The type is resolved at runtime.

Difference Between var and dynamic

|  |  |  |
| --- | --- | --- |
| **Feature** | **var** | **dynamic** |
| Type Resolution | Compile-time | Runtime |
| Type Safety | Yes (Strongly typed) | No (Can change at runtime) |
| IntelliSense Support | Yes | No |
| Error Detection | Compile-time | Runtime |

**ORM tool**

ORM Lite is a lightweight, high-performance Object Relational Mapping (ORM) library for .NET. It allows developers to work with databases using POCOs (Plain Old CLR Objects) without writing extensive SQL queries.

Key Features:

- Simple API for CRUD operations.

- Supports multiple databases like MySQL, SQL Server, and SQLite.

- Lightweight and easy to integrate.

**Install ORMLite NuGet Package :**

ServiceStack

ServiceStack.OrmLite

ServiceStack.OrmLite.MySql

**Connection string in Web.config**

<connectionStrings>

<add name="DefaultConnection" connectionString="Server=localhost;Database=BookStoreORM;User ID=Admin;Password=gs@123;SslMode=None;" providerName="MySql.Data.MySqlClient" />

</connectionStrings>

**Configuration in Global.asax**  
 var connectionString = ConfigurationManager.ConnectionStrings["DefaultConnection"].ConnectionString;

var dbFactory = new OrmLiteConnectionFactory(connectionString, MySqlDialect.Provider);

Application["DbFactory"] = dbFactory;

**Use DbFactory in BL classes**

\_dbFactory = HttpContext.Current.Application["DbFactory"] as IDbConnectionFactory;

var db = \_dbFactory.OpenDbConnection()

now we can use OrmLite methods using “db”.