1.

Memory Management

Class – Objects of type class are always stored on the **Heap**. Memory management is handled by the **Garbage Collector**, which frees objects when they are no longer referenced. A variable of type class does not hold the data itself but a **reference** (a pointer) to the memory address of the object. Therefore, if two variables point to the same object, a change made through one variable will also affect the other.

Struct – Objects of type struct are usually stored on the **Stack** when defined as local variables, and they are released automatically when the scope ends. When a struct is a field inside a class, it is stored as part of the object and therefore located on the Heap. A struct variable holds the data itself, so copying it creates a new independent copy. A change in one copy does not affect the other.

Inheritance

Class – A class can inherit from **one base class** and automatically get its fields and methods. It can extend them or override their behavior. A class can also implement **multiple interfaces**, which define methods that must be implemented. Methods in classes can be declared as virtual or abstract, which enables strong polymorphism.

Struct – A struct cannot inherit from another struct or class. By default, all structs inherit from System.ValueType, which gives them basic methods like Equals(), GetHashCode(), and ToString(). A struct can implement interfaces, but it cannot contain virtual or abstract methods because it does not support inheritance.

Constructors

Class – A class can have any kind of constructors. If no constructor is defined, the compiler provides a default empty constructor. Constructors with parameters can be defined (constructor overloading), allowing objects to be created in different ways. Static constructors can also be defined and run only once per class. An instance of a class is always created with the keyword new.

Struct – A struct always has a default constructor that initializes all fields to their default values (0, null, etc.). It is not possible to manually define a parameterless constructor, but constructors with parameters and static constructors can be defined. Unlike classes, a struct can be created without using the new keyword, but in that case all fields must be initialized manually before use.

Nullability

Class – Variables of type class can hold the value null, which means they do not point to any object. Trying to use a method or field in this state will cause a NullReferenceException. Therefore, an object of type class should always be checked for null before accessing its methods or fields.

Struct – Variables of type struct cannot be null by default , they always contain a value. If a “null” value is needed, you can use a nullable type by adding a question mark: int? number = null;

Destructor

Class – A class can have a destructor. The Garbage Collector releases memory of the object, but if the object holds external resources (like an open file or a database connection), the destructor is used to release them.

Struct – A struct cannot have a destructor, and it doesn’t need one. Once it goes out of scope, its data is removed automatically. Structs are not designed to hold complex external resources.

Common Use Cases

Class – Best used for representing complex objects that have identity, multiple fields, and their own rules or operations. Classes support inheritance, polymorphism, and flexible management, so they are used in most cases.

Struct – Best for small, simple data that represents a value, without inheritance or complex logic. Structs can be more efficient than classes because the data is stored directly in the variable, avoiding the indirection of a reference and reducing the work of the Garbage Collector. However, this is mainly true for small and simple data, since copying large structs can reduce performance.

2.

Try-Catch-Finally

This is a mechanism for handling exceptions. Inside the try block, you write code that might throw an exception. If an exception happens, the program jumps to the catch block where it can be handled. At the end, an optional finally block runs, whether an exception happened or not. Its main use is to release resources, like closing files or network connections.

1. The try block runs first.
    - If no exception happens → go directly to finally (if defined).
    - If an exception happens → jump to the matching catch.
2. The catch block runs (if defined).
3. The finally block runs last, always, even if there is a return inside the try.

It should be used when running code that might fail at runtime, such as file operations, database access, or network communication. The catch block allows handling the error gracefully, and the finally block ensures that resources are always released.

Select – Used to create a new collection from an existing one by mapping each element into a different form. The mapping is performed using a function, most commonly expressed as a lambda expression, which takes an element as input and returns a transformed value.

The result of this operation is an object of type IEnumerable<T>, representing a sequence of elements of the new type that has been generated.

To actually work with the results, one can:

- Iterate over the elements with a foreach loop, or
- Convert the sequence into a concrete collection in memory using ToList() or ToArray().

Where – Used to create a new collection from an existing one by filtering its elements according to a given condition (Predicate). The filtering is performed using a function, most commonly expressed as a lambda expression, which takes an element as input and returns a Boolean value: true if the element should be included in the result and false otherwise.

The result of this operation is an object of type IEnumerable<T>, representing a sequence of elements of the original type, but restricted to those that satisfy the filtering condition.

To actually work with the results, one can:

- Iterate over the elements with a foreach loop, or
- Convert the sequence into a concrete collection in memory using ToList() or ToArray().

FirstOrDefault – Used to return the first element from a collection. It can be applied directly to the collection, or supplied with a Predicate function in order to return the first element that satisfies a given condition.

If a matching element exists, its value is returned. If no such element is found, the method returns the default value (default).

The result of the operation is a single value of type T.

To safely use the result, the value should be stored in a variable and checked against default (for value types) or null (for reference types) before further use.

Any – Used to check whether a collection contains elements. It can operate without a condition, in which case it returns true if there is at least one element in the collection and false if the collection is empty. Alternatively, a Predicate function can be provided, in which case it returns true if at least one element satisfies the condition and false if no matching element is found.

The result of this operation is a Boolean value (bool).

The method is efficient because it stops as soon as a matching element is found, which makes it preferable in many cases to using Count() > 0.

4.

Encapsulation

Encapsulation means hiding the internal implementation of an object and only allowing access through its methods. From the outside, the object is seen as a “black box,” and interaction is done only through its public interface. This ensures control over the data, keeps it consistent, and allows the internal code to change without affecting external code.

Abstraction

Abstraction means showing only what the object can do and hiding how it does it. The user focuses on the external abilities without dealing with technical details. This principle makes code simpler to use and easier to maintain.

Inheritance

Inheritance allows creating a new class based on an existing one. The child class automatically receives the fields and methods of the parent class and can use them, extend them, or override them. This avoids code duplication, builds logical hierarchies, and provides the foundation for polymorphism.

Polymorphism

Polymorphism means that the same action can behave differently depending on the object that performs it. A base class defines a general method, and child classes override it with their own specific behavior. This allows calling the same method on different objects and getting results appropriate to each one.

Polymorphism = same method, different behavior, enabled by inheritance.