◆ 1. Can we use keywords as an identifier? Why?

No, we **cannot** use keywords as identifiers in programming languages like C#, Java, or Python.

- **Why?** Keywords are **reserved words** that have special meaning in the language's syntax (e.g., `class`, `if`, `for`, `while`). Using them as identifiers (like variable names, function names) would create ambiguity in code.

◆ 2. Differentiate explicit and implicit conversation (likely meant "conversion")

Aspect	Implicit Conversion	Explicit Conversion (Casting)
Performed by	Compiler	Programmer
Syntax	Automatic	Requires cast operator e.g., `(int)3.14`
Data Loss	No	Possible
Example	`int x = 10; double y = x;`	`double y = 9.8; int x = (int)y;`

◆ 3. Write a program to perform an example of data hiding

Data hiding is achieved using access modifiers like `private`.

```
using System;

class Person
{
    private string name; // data hiding

    public void SetName(string newName)
    {
        name = newName;
    }

    public void DisplayName()
    {
        Console.WriteLine("Name: " + name);
    }
}

class Program
{
    static void Main()
    {
        Person p = new Person();
        p.SetName("John");
        p.DisplayName();
    }
}
```

◆ 4. How can we manage runtime errors?

Runtime errors are managed using **exception handling** with `try-catch-finally`.

```
try
{
    int x = 10 / 0; // will throw DivideByZeroException
}
catch (DivideByZeroException ex)
{
    Console.WriteLine("Cannot divide by zero.");
}
finally
{
    Console.WriteLine("This always executes.");
}
```

◆ 5. What is abstract class?

An **abstract class** is a class that **cannot be instantiated** and may contain **abstract methods** (without body) that **must be overridden** in derived classes.

```
abstract class Animal
{
    public abstract void MakeSound();
}

class Dog : Animal
{
    public override void MakeSound()
    {
        Console.WriteLine("Bark");
    }
}
```

◆ 6. What is DLL?

DLL (Dynamic Link Library) is a file that contains compiled code and resources that can be used by multiple programs simultaneously.

- Allows **code reuse**, **modularity**, and **faster program execution**.
- File extension is usually `.dll`.

◆ 7. What is Namespace?

A **namespace** is a container that holds classes, interfaces, functions, etc., and helps organize code and avoid **naming conflicts**.

```
namespace MyApp.Utilities
{
    class MathHelper { }
}
```

Usage:

```
using MyApp.Utilities;
```

◆ 8. Difference between `for` loop and `foreach` loop

Feature	for loop	foreach loop
Control	Full control over loop (index, step)	Simplified iteration over collection
Use Case	When index is needed or modified	When reading elements in a collection
Example	`for(int i = 0; i < arr.Length; i++)`	`foreach(var item in arr)`

◆ 9. What is Dictionary? Advantages of Dictionary?

A **Dictionary** is a **collection of key-value pairs** in C#.

```
Dictionary<string, int> ages = new Dictionary<string, int>();
ages["Alice"] = 25;
```

Advantages:

- **Fast lookup** by key (O(1) average time).
- No duplicate keys allowed.
- Ideal for mapping relationships (e.g., name → age).

◆ 10. How to prevent a class from being instantiated?

You can make a class **abstract** or use a **private constructor** to prevent instantiation.

Using `abstract` class:

```
abstract class Utility { }
```

Using private constructor:

```
class Utility
{
    private Utility() { }
}
```

You cannot generally use **keywords** as identifiers in C#, but there is a specific exception.

Keywords as an Identifier

No, you generally **cannot use keywords as identifiers** (variable names, method names, etc.) in C#.

Why?

Keywords are **reserved words** that have a predefined, special meaning to the C# compiler (e.g., `class`, `public`, `if`, `return`). If you used them as an identifier, the compiler wouldn't be able to distinguish between your custom name and the language's built-in functionality, causing a **syntax error**.

The Exception: The `@` Prefix

C# allows you to use a keyword as an identifier if you prefix it with the **verbatim identifier symbol** (`@`). This tells the compiler to treat the following word as a regular identifier, even if it's a reserved keyword.

Example:

C#

```
int @class = 10; // Valid
Console.WriteLine(@class);
```

Explicit and Implicit Conversion

In C#, conversion (or casting) is how a value of one data type is transformed into a value of another.

Feature	Implicit Conversion	Explicit Conversion (Casting)
Definition	Performed **automatically** by the C# compiler without special syntax.	Requires **explicit syntax** (a cast operator) by the programmer.
Data Loss	**Safe** and **lossless**. Occurs when converting from a smaller type to a larger type.	**Possible** (or likely). Occurs when converting from a larger type to a smaller type.
Example	`int i = 5; double d = i;` (The `int` fits safely into the `double`.)	`double d = 5.7; int i = (int)d;` (The programmer forces the conversion, losing the `.7`.)

Export to Sheets

Program Example of Data Hiding (C#)

Data Hiding is achieved in C# by using the **`private`** access modifier, which restricts direct access to a class's internal fields from outside the class. Access is only allowed through controlled public methods (getters and setters).

C#

```
using System;

public class BankAccount
{
    // The account balance is 'hidden' using the private access modifier.
    // It can only be accessed or modified within this class.
    private decimal accountBalance = 0;

    public BankAccount(decimal initialBalance)
    {
        if (initialBalance >= 0)
        {
            accountBalance = initialBalance;
        }
    }

    // Public method to allow controlled modification (Deposit)
    public void Deposit(decimal amount)
    {
        if (amount > 0)
        {
            accountBalance += amount;
            Console.WriteLine($"Deposited: ${amount}.");
        }
    }

    // Public method to allow controlled access (Getter)
    public decimal GetBalance()
    {
        return accountBalance;
    }
}

public class Program
{
    public static void Main()
    {
        BankAccount account = new BankAccount(500.00m);
        // Intended, controlled access via a public method
        Console.WriteLine($"Current Balance: ${account.GetBalance()}");

        account.Deposit(150.50m);
        Console.WriteLine($"New Balance: ${account.GetBalance()}");

        // The following line would cause a compiler error, demonstrating 
data hiding:
        // account.accountBalance = 9999999.00m;
    }
}
```

How Can We Manage Runtime Errors?

Runtime errors (or **exceptions**) are managed in C# primarily through **Exception Handling** using the **`try-catch-finally`** block structure:

1. **`try`:** Encloses the block of code that might generate an exception (e.g., file operations, network calls, division by zero).
2. **`catch`:** Executes if an exception occurs within the `try` block. It allows you to **gracefully handle** the error (e.g., log the error, display a user-friendly message, or provide a default value) instead of crashing the program. You can specify different `catch` blocks for different exception types.
3. **`finally`:** Executes **regardless** of whether an exception occurred or was handled. It is used for **cleanup code**, such as closing file streams or database connections, ensuring resources are released.

What is an Abstract Class?

An **abstract class** is a class designed to be **inherited from** (a base class) but **cannot be instantiated** directly.

- It acts as a **blueprint** or **contract** for its derived classes.
- It can contain **abstract methods** (methods with signatures but **no body**), which **must be implemented** by any non-abstract derived class.
- It can also contain standard (concrete) methods and fields with full implementation.
- It is declared using the **`abstract`** keyword.

What is a DLL?

DLL stands for **Dynamic Link Library**.

- A DLL is a file that contains code and data that can be used by multiple programs simultaneously.
- It allows for **code reuse** and **modularity**. For example, the system libraries that handle graphics or file I/O are typically stored in DLLs.
- The term "Dynamic Link" means that the application links to the necessary code **at runtime** (when the program is executed), rather than compiling all the code into the main executable file.

What is a Namespace?

A **namespace** is a container that provides a way to organize C# code and prevent **naming collisions**.

- It groups related types (classes, structs, interfaces, etc.) under a unique, hierarchical name (e.g., `System.Collections.Generic`).
- Without namespaces, if two different libraries both defined a class called `Logger`, using both libraries would cause an ambiguous reference error.
- Namespaces solve this by allowing you to refer to them as `LibraryA.Logger` and `LibraryB.Logger`. You use the **`using`** keyword to import a namespace and simplify type names.

Difference Between `for` Loop and `foreach` Loop

Feature	`for` Loop	`foreach` Loop
Purpose	To iterate a block of code a **specific number of times** or to access elements using an **index**.	To iterate through **every item** in a collection (array, list, etc.).
Mechanism	Uses an **explicit counter/index** variable (`i`) which you manage (initialize, condition, increment).	**Iterates directly** over the elements in the collection; the index is **not exposed**.
Modification	Allows **modification** of the collection's contents and the ability to skip or jump iterations using `continue` or `break`.	Does **not allow modification** (adding or removing elements) of the collection structure during iteration.
Best Use	When you need the **index** (e.g., accessing two arrays simultaneously) or when iterating **backwards**.	For simple, **safe iteration** over all items, which is generally more readable.

Export to Sheets

What is Dictionary? Advantages of Dictionary?

What is a Dictionary?

In C#, a `Dictionary<TKey, TValue>` is a generic collection that stores data as a set of **key-value pairs**. It is conceptually similar to an associative array or hash map.

- The **key** must be unique and is used to retrieve the associated **value**.
- The implementation uses **hashing** to provide extremely fast lookups.

Advantages of Dictionary

1. **Fast Lookup (O(1) Average Time):** The primary advantage is its efficiency. Retrieving a value based on its key is done in near-constant time, making it much faster than searching a `List` or array.
2. **Intuitive Data Association:** It provides a clear, logical way to link two pieces of data (e.g., mapping a **UserID** to a **UserObject**).
3. **Unique Keys:** It guarantees that every key in the collection is unique, preventing redundant or conflicting entries.

How to Prevent a Class from Being Instantiated?

You can prevent a class from being instantiated (i.e., prevent objects from being created with `new ClassName()`) by making its **constructor private**.

This technique is primarily used for:

1. **Static Utility Classes:** Classes that only contain static methods and properties (e.g., a `Math` utility class) where no instance data is needed.
2. **Singleton Pattern:** Where you need to strictly ensure that **only one instance** of the class exists in the entire application.

C#

```
public class Singleton
{
    // Make the constructor private to block external instantiation
    private Singleton()
    {
        // Initialization code here
    }

    // Static instance of the class (the single instance allowed)
    private static Singleton instance = new Singleton();

    // Public static method to provide the single instance
    public static Singleton GetInstance()
    {
        return instance;
    }
}

// Usage in another class:
// Singleton s = new Singleton(); // COMPILER ERROR: Inaccessible due to
protection level
Singleton s = Singleton.GetInstance(); // Correct and only way to get the
object
```

1. What is the .NET Framework / .NET Core?

The **.NET Framework** is Microsoft's original software framework primarily for Windows applications.

.NET Core (now simply **.NET**) is the modern, open-source, and **cross-platform** (Windows, Linux, macOS) successor. It's designed for developing modern applications like cloud, web, and mobile apps.

---

2. What is the difference between `const` and `readonly`?

Both are used to define fields whose values cannot be changed, but they differ in when the value is assigned:

Feature	`const` (Constant)	`readonly`
Assignment Time	**Compile-time** only. Must be assigned when declared.	**Run-time** (either at declaration or within the constructor).

Feature	`const` (Constant)	`readonly`
Type Restriction	Can only be used on value types or `string`.	Can be used on **any type**.
Scope	Always **implicitly static**.	Can be **instance** or **static**.

Export to Sheets

Example:

C#

```
public class Settings {
    public const int MaxUsers = 100; // Assigned at compile time
    public readonly DateTime StartTime; // Assigned in the constructor

    public Settings() {
        StartTime = DateTime.Now; // Assigned at runtime
    }
}
```

---

3. What are the four main pillars of OOP in C#?

C# is an object-oriented programming (OOP) language built on these four concepts:

1. **Encapsulation:** Bundling data (fields) and methods that operate on that data into a single unit (class). It's primarily achieved using **access modifiers** (`public`, `private`, etc.) to **hide** the internal state.
2. **Inheritance:** The mechanism by which one class (**derived class**) acquires the properties and methods of another class (**base class**), promoting code reuse.
3. **Polymorphism:** The ability of an object to take on many forms. It allows an action to be performed in different ways, often achieved through **method overriding** (run-time) and **method overloading** (compile-time).

4. **Abstraction:** Hiding the complex implementation details and showing only the necessary features of the object. This is achieved using **abstract classes** and **interfaces**.

---

4. Explain the difference between Value Types and Reference Types.

Feature	Value Types	Reference Types
Storage	Stored on the **Stack**.	Stored on the **Heap**.
Data Storage	Stores the **actual data** directly.	Stores a **reference (memory address)** to the data.
Assignment	A new copy of the value is made.	Only the reference (address) is copied; both variables point to the same data.
Examples	`int`, `char`, `float`, `struct`, `enum`.	`class`, `string`, `interface`, `array`, `delegate`.

Export to Sheets

---

5. What are Access Modifiers?

Access modifiers are keywords used to specify the declared **accessibility** (or scope) of a type (class), member (method, property), or data field.

Modifier	Accessibility
`public`	Accessible from **anywhere**.
`private`	Accessible **only within the defining class**. (Default for class members)
`protected`	Accessible only within the defining class and by **derived classes**.
`internal`	Accessible only within the **same assembly** (project). (Default for classes)

Modifier	Accessibility
`protected internal`	Accessible within the current assembly **OR** by derived classes in any assembly.
`private protected`	Accessible within the current assembly **AND** by derived classes in the same assembly.

Export to Sheets

---

6. What is an Interface?

An **interface** is a contract defined by a set of related **public** members (methods, properties, events, indexers).

- It contains **only declarations**, no implementation code.
- A class that implements an interface **must** provide concrete implementation for **all** of its members.
- A class can inherit from only one base class but can implement **multiple interfaces**, which is C#'s way of achieving multiple inheritance of behavior.

---

7. What is the `using` statement?

The `using` statement (specifically the *resource management* form) is a convenience feature that ensures that an object implementing the **`IDisposable`** interface is **disposed of correctly** when it goes out of scope, even if an exception occurs.

It's primarily used for resources that hold external data like file handles, database connections, or network sockets.

C#

```
// The connection object implements IDisposable
```

```
using (SqlConnection conn = new SqlConnection("..."))
{
    conn.Open();
    // Do work...
} // conn.Dispose() is automatically called here, closing the connection.
```

---

8. What is Method Overloading?

Method overloading is a form of compile-time polymorphism where multiple methods in the same class share the **same name** but have **different signatures** (i.e., they must differ in the number, type, or order of their parameters).

The return type can be the same or different, but it alone is **not enough** to overload a method.

Example:

C#

```
public int Add(int a, int b) { return a + b; }
public double Add(double a, double b) { return a + b; }
public int Add(int a, int b, int c) { return a + b + c; }
```

---

9. What is the purpose of `static`?

The `static` keyword is used to declare members that belong to the **class itself** rather than to any specific instance of the class.

- **Static members** are accessed using the class name, not an object name (e.g., `Math.Sqrt(4)`).
- **Static classes** can only contain static members and cannot be instantiated.
- Static constructors are used to initialize static fields or to perform a particular action only once.

---

10. What is an Assembly?

In C# and .NET, an **Assembly** is the fundamental unit of deployment, versioning, reuse, and security.

- It is compiled code (Intermediate Language or **IL**) that is packaged into a file, typically with a **`.dll`** (Dynamic Link Library) or **`.exe`** (executable) extension.
- Every C# project that you compile generates an assembly.
- An assembly contains a **Manifest**—a table of contents that describes everything in the assembly, including its version, dependencies, and files.

1. What is the CLR (Common Language Runtime)?

The **CLR** is the virtual machine component of the .NET Framework and .NET Core. It's the runtime environment that manages the execution of .NET programs.

- It performs key services like **Just-In-Time (JIT) compilation**, **memory management** (Garbage Collection), **security checks**, and **exception handling**.
- The CLR allows code written in different .NET languages (C#, F#, VB.NET) to interoperate, as they all compile down to a common intermediate format called **CIL (Common Intermediate Language)**.

---

2. Explain Boxing and Unboxing.

Boxing and **Unboxing** are processes that bridge the gap between **value types** (stored on the Stack) and the **object type** (stored on the Heap).

Process	Direction	What Happens	Performance
Boxing	**Value Type →** **Reference Type** (e.g., `int` → `object`)	The value is copied from the Stack to the Heap.	Implicit, but **slows performance** due to allocation and copying.

Process	Direction	What Happens	Performance
Unboxing	**Reference Type → Value Type** (e.g., `object` → `int`)	The value is copied from the Heap back to the Stack.	Explicit (requires a cast), also **slow** and can throw a **`InvalidCastException`** if the type doesn't match.

Export to Sheets

Example:

C#

```
int i = 10;
object o = i;  // Boxing: 'i' is copied to the Heap
int j = (int)o; // Unboxing: 'o' is copied back to the Stack
```

---

3. What is the Garbage Collector (GC)?

The **Garbage Collector (GC)** is the automatic memory management system in the CLR. Its primary job is to **reclaim memory** occupied by objects that are no longer being used by the application.

- The GC automatically tracks objects on the **Heap**.
- When an object is unreachable (no variables are referencing it), the GC marks it as garbage and eventually **deallocates** the memory.
- This prevents memory leaks and makes C# programming easier, as developers don't have to manually free memory (unlike C++).

---

4. What are Delegates?

A **Delegate** is a **type-safe function pointer**. It holds a reference to one or more methods.

- It defines the **signature** (return type and parameters) of the methods it can point to.
- Delegates are the foundation for **events** and **callbacks** in C#.

- They allow methods to be passed as arguments to other methods, enabling concepts like event handling and asynchronous programming.

Multicast Delegate: A delegate instance can hold references to multiple methods. When the delegate is invoked, all referenced methods are called sequentially.

---

5. Explain `params` and when it's useful.

The **`params`** keyword allows a method to accept a **variable number of arguments** of a specific type.

- It must be the **last parameter** in the method's parameter list.
- The argument type must be a **single-dimensional array**.

It's useful for creating methods where you don't know the exact number of inputs the caller will provide (e.g., a utility function that sums numbers or prints a list of items).

Example:

C#

```
public int CalculateSum(params int[] numbers)
{
    int sum = 0;
    foreach (int n in numbers)
    {
        sum += n;
    }
    return sum;
}

// Usage:
int total1 = CalculateSum(1, 2, 3); // Passes 3 arguments
int total2 = CalculateSum(10, 20, 30, 40, 50); // Passes 5 arguments
```

---

6. What is the difference between an Array and an ArrayList?

Feature	Array (`int[]`, `string[]`)	ArrayList
Type	Value or Reference Type	Reference Type (Class in `System.Collections`)
Size	**Fixed** size. Cannot be resized after creation.	**Dynamic** size. Automatically resizes as elements are added or removed.
Type Safety	**Type-safe**. Can only store its declared type (e.g., only `int`s).	**Not type-safe**. Stores elements as `object` (requires boxing/unboxing).
Performance	**Better** performance due to no boxing/unboxing overhead.	**Slower** performance due to boxing/unboxing and resizing operations.
Modern Alternative	N/A	**`List<T>`** (Generic List)

Export to Sheets

---

7. What is an Indexer?

An **indexer** allows instances of a class or struct to be accessed just like an **array**. It lets you retrieve or set values using square brackets (`[]`).

- Indexers are essentially a special type of property that takes one or more arguments (the index).
- They provide a more intuitive way to access data internally managed by a class, like a collection or dictionary.

Example:

C#

```
public class MyCollection
```

```
{
    private string[] data = new string[10];

    // Indexer definition
    public string this[int index]
    {
        get { return data[index]; }
        set { data[index] = value; }
    }
}

// Usage:
MyCollection col = new MyCollection();
col[0] = "First Item"; // Calls the set accessor
string item = col[0];  // Calls the get accessor
```

---

8. What is the `StringBuilder` class and why is it used?

The `StringBuilder` class is located in the `System.Text` namespace and is used for **efficiently manipulating strings**.

- C# strings are **immutable** (cannot be changed after creation). Every time you modify a `string` (e.g., using `+` or concatenation), a brand new string object is created in memory.
- `StringBuilder` is **mutable**. When you append or modify a `StringBuilder` object, it usually modifies the string *in place* without creating a new object for every change.

It is best used when performing a **large number of string concatenations** in a loop, as it significantly reduces memory allocation overhead and improves performance.

---

9. Explain the `throw` and `throws` keywords.

Keyword	Purpose	Usage in C#
`throw`	Used to **explicitly raise** an exception during program execution.	Used inside a `try` block or method to signal an error condition.
`throws`	Used in a method **signature** to declare that the method *might* throw an exception.	**Does not exist in C#.** This keyword is part of Java's checked exception mechanism. C# only uses **`throw`**.

Export to Sheets

---

10. What is a Generic?

Generics allow you to define classes, methods, and interfaces that operate on a **specified data type**, without committing to that specific type until it is instantiated or used. The type is specified using a placeholder, typically denoted by **`<T>`** (for Type).

Advantages:

1. **Type Safety:** Eliminates the need for boxing/unboxing and the risk of runtime `InvalidCastException`s associated with non-generic collections like `ArrayList`.
2. **Performance:** Since there's no boxing/unboxing to the `object` type, performance is significantly better.
3. **Code Reuse:** You can write a single class (like `List<T>`) that works with any data type.

Example:

C#

```
// The generic List<T>
List<int> numbers = new List<int>();
numbers.Add(10); // Only accepts integers

List<string> names = new List<string>();
```

```
names.Add("Alice"); // Only accepts strings
```

Certainly! Here are ten more advanced C# interview questions, touching upon asynchronous programming, LINQ, and advanced OOP concepts.

---

1. Explain the Async/Await Pattern.

Async/Await is C#'s primary pattern for writing **asynchronous** code that is as readable as synchronous code. It's used to prevent the application from blocking while waiting for a long-running operation (like a network request, file I/O, or database query) to complete.

- **`async` keyword:** Used to modify a method, signaling that it may contain asynchronous operations. An `async` method usually returns a `Task` or `Task<T>`.
- **`await` keyword:** Used inside an `async` method to **pause execution** until the awaited asynchronous task completes. Crucially, the **thread is freed** to handle other work during this waiting time, improving application responsiveness, especially in UI and server applications.

Example:

C#

```
public async Task<string> FetchDataAsync() {
    // The thread is returned to the thread pool while the database operation
runs.
    var data = await dbContext.Users.ToListAsync();
    return data.ToString();
}
```

---

2. What is LINQ?

LINQ stands for **Language Integrated Query**. It is a uniform programming model added to C# and .NET for querying data from various sources (databases, XML documents, in-memory collections like arrays and lists) using a syntax similar to SQL.

- **Uniformity:** It allows developers to use the **same syntax** (`from...where...select`) regardless of the data source.
- **Type Safety:** Queries are checked at **compile time**, unlike string-based SQL queries, catching errors earlier.
- **Readability:** It makes code querying data much more readable and maintainable.

---

3. Differentiate between `IEnumerable<T>` and `IQueryable<T>`.

Both are interfaces used for querying, but they handle query execution differently:

Feature	`IEnumerable<T>`	`IQueryable<T>`
Execution	**In-memory** (Client-side) filtering/processing.	**Database** (Server-side) filtering/processing.
Location	Executes after **all** data is loaded into application memory.	Executes before retrieving the data; it builds an **Expression Tree**.
Use Case	Ideal for **in-memory collections** (arrays, lists) or small data sets.	Essential for **external data sources** (like databases) to prevent over-fetching.
Efficiency	Less efficient for large remote data sources, as all data is downloaded first.	Highly efficient for remote data, as it translates the query to optimal SQL.

Export to Sheets

---

4. What is the difference between `is` and `as` operators?

Both are used for type checking and conversion, but they have different behaviors:

Operator	Purpose	Behavior	Failure Result
`is`	**Checks** if an object is compatible with a given type.	Returns **`true`** or **`false`**.	Never throws an exception.
`as`	Attempts a **conversion** (cast) without throwing an exception.	Returns the converted object reference.	Returns **`null`** if the conversion fails.

Export to Sheets

Example:

C#

```
object obj = "hello";

// Using 'is': Check and then cast (redundant type checking)
if (obj is string) {
    string s = (string)obj;
}

// Using 'as': Cleaner attempt (no exception on failure)
string s = obj as string; // s will be "hello"
object fail = 10;
string s_fail = fail as string; // s_fail will be null
```

---

5. What are C# Properties and how do they differ from Fields?

A **Field** is a variable that stores data directly within a class.

A **Property** is a member that provides a flexible mechanism to **read, write, or compute** the value of a private field. Properties are an extension of the concept of encapsulation, allowing controlled access to data via **accessors** (`get` and `set`).

Feature	Field	Property
Access	Direct access (often kept `private`).	Controlled access via `get` and `set` accessors.
Logic	Pure storage; cannot contain custom logic.	Can contain custom logic (validation, logging, computation).
External Use	Should generally be kept `private`.	Used publicly; accessed externally like a field, but acts like a method internally.

Export to Sheets

C# also supports **Auto-Implemented Properties** for simple cases: `public int Id { get; set; }`

---

6. Define and differentiate `sealed` and `static` classes.

Feature	`sealed` Class	`static` Class
Instantiation	Can be instantiated (objects can be created).	**Cannot** be instantiated (no objects can be created).
Inheritance	**Cannot** be inherited from. It prevents derived classes.	**Cannot** be inherited from (implicitly sealed).
Members	Can contain both instance and static members.	Can **only** contain static members.
Purpose	To prevent polymorphism and ensure the class implementation is final.	To hold utility methods or global data that doesn't rely on object state.

Export to Sheets

---

7. What is the `yield` keyword?

The **`yield`** keyword is used in C# to create an **iterator** (a method that returns an `IEnumerable` or `IEnumerator`) that performs **deferred execution** and **state retention**.

- When a `yield return` statement is encountered, the item is returned, and the current state of the iterator is saved.
- Execution resumes from the point where it left off the next time the iterator is called.
- This is known as **lazy loading** or **streaming** and is extremely memory efficient, as it avoids loading an entire collection into memory at once.

---

8. What is a Custom Exception?

A **Custom Exception** is a class you create that inherits from a standard exception class (usually `System.Exception` or `System.ApplicationException`).

- **Purpose:** To make your code more robust and readable by defining **specific error types** unique to your application's business logic.
- It allows you to provide clearer information about what went wrong and lets `catch` blocks handle errors more precisely.

Example:

C#

```
public class InsufficientFundsException : Exception {
    public InsufficientFundsException(string message) : base(message) { }
}
```

---

9. What is Reflection?

Reflection is the process of examining the **metadata** of types (classes, structs, interfaces) in an assembly at **runtime**.

- It allows a program to inspect its own structure (or the structure of external libraries).
- Using reflection, you can:
    - Inspect a class's members (methods, properties, fields).
    - Dynamically **invoke** methods or access properties by name.
    - **Create instances** of types without knowing the type at compile time.

Reflection is a key tool in serialization, dependency injection, and creating plugin architectures, but it can be slower than normal direct code execution.

---

10. Explain the concept of Covariance and Contravariance.

These concepts relate to the ability to use a more derived type (less specific) or a less derived type (more specific) than originally specified by a generic type parameter. They are crucial for generic **interfaces** and **delegates**.

1. **Covariance (`out` keyword):** Allows a method to return a **more derived type** than that specified by the generic parameter. (e.g., An `IEnumerable<string>` can be assigned to an `IEnumerable<object>`).
    - **Direction: Output** position.
2. **Contravariance (`in` keyword):** Allows a method to accept a **less derived type** than that specified by the generic parameter. (e.g., An `Action<object>` can be assigned to an `Action<string>`).
    - **Direction: Input** position.

I understand you're looking for more basic C# interview questions. Here are ten additional fundamental topics that are often tested in a C# interview.

---

1. What is the difference between an `out` parameter and a `ref` parameter?

Both keywords allow a method to change the value of an argument passed to it, but they differ in initialization:

Feature	`out` Parameter	`ref` Parameter
Initialization	The argument **doesn't need to be initialized** before being passed to the method.	The argument **must be initialized** before being passed to the method.
Assignment	The method **must assign a value** to the parameter before returning.	The method is **not required** to assign a value to the parameter.
Purpose	Used when a method needs to return multiple values.	Used when a method needs to read and possibly modify an existing value.

Export to Sheets

Key takeaway: With `out`, data flows **out** of the method; with `ref`, data flows **in and out**.

---

2. What is method overriding?

Method overriding is a feature of **run-time polymorphism** that allows a **derived class** (child class) to provide a specific implementation for a method that is already defined in its **base class** (parent class).

- The method signature (name, return type, and parameters) **must be identical**.
- The base class method must be marked with the **`virtual`** keyword.
- The derived class method must be marked with the **`override`** keyword.

This ensures that when a derived class object is treated as its base class type, the correct specialized method is executed at runtime.

---

3. What is the `base` keyword?

The **`base`** keyword is used inside a **derived class** to access members of its immediate **base class**.

Its two primary uses are:

1. **Accessing Overridden Members:** Calling a method or property from the base class that was overridden in the derived class.
2. **Calling a Base Class Constructor:** Used within the derived class constructor to explicitly call a specific constructor of the base class.

Example (Calling Base Constructor):

C#

```
public class BaseClass {
    public BaseClass(string name) { /* ... */ }
}
public class DerivedClass : BaseClass {
    public DerivedClass(string name) : base(name) { }
}
```

---

4. What is a Struct?

A **Struct** (`structure`) in C# is a **value type** (unlike classes, which are reference types) primarily used to hold a small group of related variables.

- Structs are best used for **lightweight objects** that represent single values (like coordinates, currency amounts, or `DateTime`).
- They are stored on the **Stack**, making memory allocation fast.
- They **cannot be inherited** (implicitly sealed), and they **do not support explicit parameterless constructors**.

---

5. What is the difference between a `throw` and a `throw ex`?

Both statements re-throw an exception, but their behavior regarding the **stack trace** is critical:

1. **`throw` (Recommended):** Re-throws the exception while preserving the **original stack trace**. This is crucial for debugging, as it tells you the original source of the error.
2. **`throw ex` (Avoid):** Re-throws the exception but resets the stack trace to the point where `throw ex` is called. This obscures the original location of the error, making debugging much harder.

Always use `throw` by itself within a `catch` block to re-throw an exception and preserve the stack trace.

---

6. What is the difference between `checked` and `unchecked`?

These C# keywords control the compiler's behavior regarding **arithmetic overflow** for integer types.

- **`checked`:** Tells the runtime to explicitly check for arithmetic overflow. If an overflow occurs (e.g., adding 1 to the maximum value of an `int`), it will throw an **`OverflowException`**.
- **`unchecked`:** Tells the runtime **not** to check for overflow. If an overflow occurs, the result will simply **wrap around** (discard the most significant bits) without throwing an exception.

By default, operations in C# are **unchecked**, except for constant expression evaluations during compilation.

---

7. What is an enum?

An **enum** (short for **enumeration**) is a **value type** that defines a set of **named integral constants**.

- It is used to assign meaningful names to integral values, making code more readable and maintainable.
- The underlying type of an enum is an integer (`int`) by default, but it can be changed (e.g., to `byte`, `short`, `long`).

Example:

C#

```
public enum Status
{
    Pending = 1, // Default value starts at 0 if not specified
    Processing = 2,
    Completed = 3
}

Status orderStatus = Status.Completed;
```

---

8. Explain the difference between Early Binding and Late Binding.

These terms describe when a method call is resolved to a specific method implementation:

1. **Early Binding (Static Binding):**
   - Resolution occurs at **compile time**.
   - Used for normal method calls, overloaded methods, and static methods.
   - **Faster** because the compiler knows exactly which method to call.
2. **Late Binding (Dynamic Binding):**
   - Resolution occurs at **run time**.
   - Used with **virtual/override methods** (polymorphism) and the **`dynamic`** keyword.

- **Slower** because the method to be called must be determined by the CLR during execution.

---

9. What is a `string` versus a `StringBuilder`?

Feature	`string`	`StringBuilder`
Mutability	**Immutable** (cannot be changed after creation).	**Mutable** (can be changed in place).
Concatenation	Creates a **new string object** in memory for every modification.	Modifies the internal buffer **in place** (reallocates less frequently).
Performance	**Poor** for large numbers of concatenations (e.g., in a loop).	**Excellent** for extensive string manipulation.
Memory	High memory overhead due to frequent object creation.	Lower memory overhead for heavy modifications.

Export to Sheets

---

10. What are Nullable Types (`?`)?

A **Nullable Type** allows a **value type** (like `int`, `bool`, `DateTime`, or `struct`) to hold its normal range of values **plus** the value `null`.

- Reference types (like `string` or `class`) can always be `null`, but value types normally cannot.
- Nullable types are declared using the question mark suffix (`?`).

Example:

C#

```
// i can hold any integer value OR null
```

```
int? i = null;

// A Nullable type has two key properties:
if (i.HasValue)
{
    int value = i.Value; // Get the underlying value
}
```

1. Explain the Garbage Collection (GC) Generations in the CLR.

The CLR's Garbage Collector (GC) uses a **generational model** to optimize performance, operating under the principle that **most objects die young**. The Heap is divided into three generations:

1. **Generation 0 (Gen 0):** The youngest generation. This is where all **newly created, short-lived objects** (like local variables or method parameters) are placed. GC checks this generation most frequently.
2. **Generation 1 (Gen 1):** A buffer between the very young and the long-lived objects. Objects that **survive a Gen 0 collection** are promoted here. GC checks this less frequently than Gen 0.
3. **Generation 2 (Gen 2):** The oldest generation. Contains **long-lived objects** that survive multiple collections (like static objects, singletons, or high-level caches). GC checks this least frequently.

This model speeds up garbage collection by focusing its effort on the small area (Gen 0) where most objects are expected to die.

---

2. What is the difference between `IComparable` and `IComparer`?

Both interfaces are used for sorting in C#, but they define the comparison logic differently:

Interface	Purpose	Implementation Location	Usage
`IComparable`	Defines the **default or natural ordering** for a class.	Implemented **by the class itself** (the objects being compared).	Used by `List<T>.Sort()` when no arguments are passed.
`IComparer`	Defines a **custom comparison logic** separate from the class.	Implemented by a **separate class** (the comparer).	Used to provide **multiple, external sorting orders** (e.g., sort by name, then by date).

Export to Sheets

- `IComparable` has one method: `CompareTo(object other)`.
- `IComparer` has one method: `Compare(object x, object y)`.

---

3. What is an Extension Method?

An **Extension Method** allows you to "add" new methods to existing classes, structs, or interfaces **without** modifying the original class's source code, inheriting from it, or recompiling it.

- They are defined as `static` methods in a `static` class.
- The first parameter must be preceded by the **`this`** keyword, followed by the type you are extending.

Extension methods are commonly used to add LINQ capabilities to `IEnumerable<T>` and to make utility functions appear as if they belong to the object itself.

Example:

C#

```
// Defined in a static utility class
public static class StringExtensions
```

```
{
    public static string ToTitleCase(this string input)
    {
        // ... custom logic ...
    }
}

// Usage looks like a regular instance method
string name = "alice";
string title = name.ToTitleCase();
```

---

4. What is a Delegate Chain (Multicast Delegate)?

A **Delegate Chain** (or Multicast Delegate) is a feature where a single delegate instance can hold references to **multiple methods** that share the same signature.

- You use the `+=` operator to add a method to the chain and the `-=` operator to remove one.
- When the delegate is **invoked**, all methods in its invocation list are called sequentially, in the order they were added.
- This is the underlying mechanism for **events** in C#.

Note on Return Values: If the delegate returns a value, the delegate invocation returns the value from **only the last method** executed in the chain.

---

5. Explain the purpose of the `using static` directive.

The **`using static`** directive (introduced in C# 6) allows you to import the **static members** (methods, properties, fields) of a class directly into your code file, letting you call them without prefixing the class name.

- **Before:** `double root = Math.Sqrt(4);`
- **With `using static`:**

C#

```
using static System.Math;
// ...
double root = Sqrt(4); // No need for 'Math.' prefix
```

This is useful for utility classes like `System.Math`, `System.Console`, or custom helper classes, improving code readability when using these static members repeatedly.

---

6. What is the `Dictionary<TKey, TValue>` internal implementation based on?

The C# `Dictionary<TKey, TValue>` is internally implemented using a **hash table** (specifically, an array of buckets, where each bucket holds a linked list of entries).

1. When an item is added, the key's **`GetHashCode()`** method is called to generate a unique hash value.
2. This hash value determines the **index (bucket)** in the internal array where the entry will be stored.
3. For retrieval, the key is hashed again, and the dictionary jumps directly to the bucket index, then compares the key using **`Equals()`** to find the correct value.

This hashing mechanism is why dictionary lookups are so fast, averaging **O(1)** time complexity.

---

7. What is Covariance in Generics?

Covariance allows a generic type parameter to be replaced by a **more derived type** (a subtype).

- It is supported on **return types** of methods within generic **interfaces** and **delegates**.
- The generic parameter must be marked with the **`out`** keyword.
- **Analogy:** You can assign a container of `Cats` to a container of `Animals` because a cat is an animal.

Example:

C#

```
public interface IProducer<out T> // 'out' enables covariance
{
    T Produce();
}

// Assignment is allowed:
IProducer<Cat> catProducer = new CatProducer();
IProducer<Animal> animalProducer = catProducer; // Covariance in action
```

---

8. What is the purpose of the `IDisposable` interface and the `Dispose()` method?

The **`IDisposable`** interface is used to define a mechanism for **releasing unmanaged resources** (like file handles, database connections, or network sockets) once an object is finished with them.

- It contains a single method: **`void Dispose()`**.
- When a class implements `IDisposable`, it means the object needs explicit cleanup.
- The **`using` statement** is the standard, safe way to consume an `IDisposable` object, as it guarantees that `Dispose()` will be called, even if an exception occurs.

---

9. When should you use a `struct` over a `class`?

You should choose a **`struct`** (Value Type) over a **`class`** (Reference Type) when:

1. **The object is small:** Typically smaller than **16 bytes**.
2. **The object is logically a single value:** It represents simple data (e.g., a color, a point, a key/value pair).
3. **The object's lifetime is short:** Avoiding the Heap allocation and Garbage Collection overhead provides performance benefits.

4. **You frequently pass it by value:** You want a new copy made every time it's assigned or passed to a method.

If the object is large, represents complex behavior, or requires inheritance, a `class` is the correct choice.

---

10. What is Dependency Injection (DI)?

Dependency Injection (DI) is a design pattern used to achieve **Inversion of Control (IoC)**. Instead of a component creating its dependencies (other objects it needs), those dependencies are **injected** into it from the outside.

- **Goal:** To decouple components, making the code more flexible, testable, and maintainable.
- **Mechanism:** Typically, dependencies are passed into a class through its **constructor** (Constructor Injection), method parameters, or properties.
- **DI Container:** In real-world C# applications, a specialized library (like ASP.NET Core's built-in container, or third-party tools like Autofac) manages the creation and lifetime of these dependencies.

No problem! Here are ten more fundamental and advanced C# interview questions, focusing on generics, events, data structures, and best practices.

---

1. What are Events in C#?

Events are a special type of **delegate** used for **notification** and are the core of the Observer design pattern in C#. They allow a class (**Publisher**) to notify other classes (**Subscribers**) when something interesting happens, without needing to know which subscribers are listening.

- An event is essentially a **wrapper around a delegate** that restricts access: subscribers can only use the `+=` (subscribe) and `-=` (unsubscribe) operators.
- The standard signature for an event handler is `void HandlerName(object sender, EventArgs e)`.
- Events achieve **decoupling**; the Publisher knows only that an event occurred, not who handles it.

---

2. What is the `object` type in C#?

The **`object`** type (or `System.Object`) is the **ultimate base class** for all types in the .NET type system.

- Every type in C#, whether it's a value type (like `int`) or a reference type (like `string` or any class), inherits directly or indirectly from `object`.
- It provides fundamental methods that all objects possess, such as `ToString()`, `Equals()`, and `GetHashCode()`.
- It's used when a function needs to accept any type of data, although using **generics** is preferred for type safety.

---

3. How do you create a read-only property?

You create a read-only property by defining only the **`get` accessor** and omitting the `set` accessor. The value is typically initialized in the class's constructor or when the field is declared.

Example (Using an Auto-Implemented Property):

C#

```
public class User
{
    // Read-only property initialized in the constructor
    public int Id { get; }
    // Read-only property initialized directly
    public string SystemName { get; } = "AppServer1";

    public User(int id)
    {
        Id = id; // Can only be set here, in the constructor
    }
}
```

---

4. What is a Collection Initializer?

A **Collection Initializer** is a syntactic shortcut in C# that allows you to initialize a collection (like a `List`, `Dictionary`, or array) and add elements to it in a single statement. It improves code readability and conciseness.

Example:

C#

```
// Standard List initialization
List<string> names = new List<string>();
names.Add("Alice");
names.Add("Bob");

// Collection Initializer syntax
List<string> names = new List<string>
{
    "Alice",
    "Bob",
    "Charlie"
};
```

---

5. Explain Default Interface Methods (DIMs).

Default Interface Methods (DIMs), introduced in C# 8.0, allow you to provide a **default implementation** for a method within an interface.

- **Benefit:** They allow you to add new methods to an existing interface **without breaking backward compatibility**. Classes that implement the old version of the interface don't need to be modified immediately.
- **Limitation:** DIMs cannot access fields or instance properties that aren't defined in the interface.

---

6. What is the `dynamic` keyword?

The **`dynamic`** keyword is a type in C# that tells the compiler to bypass **compile-time type checking**. Instead, the type resolution and member access (Late Binding) are deferred until **runtime**.

- It is often used when interoperating with non-C# code (like COM objects or Python scripts) or when working with data structures whose shape is unknown until runtime.
- **Trade-off:** It provides flexibility but sacrifices type safety, as any errors (like calling a non-existent method) will only be caught at runtime, causing an exception.

---

7. Differentiate between `List<T>` and `LinkedList<T>`.

Both are generic collections, but they differ fundamentally in their internal structure and performance characteristics:

Feature	`List<T>` (Array-based)	`LinkedList<T>` (Node-based)
Internal Structure	**Dynamic Array**. Stores elements contiguously in memory.	**Doubly Linked List**. Stores elements as nodes, each pointing to the next and previous node.
Element Access	**Fast** (O(1) average) via index (e.g., `list[5]`).	**Slow** (O(n)) via traversal.
Insertion/Deletion	**Slow** (O(n)) in the middle, as all subsequent elements must be shifted.	**Fast** (O(1)) once the insertion point is found.
Memory	Low memory overhead per element.	High memory overhead due to storing two pointers per node.

Export to Sheets

---

8. What is Dependency Inversion Principle (DIP)?

The **Dependency Inversion Principle (DIP)** is the "D" in the SOLID design principles. It states:

1. **High-level modules should not depend on low-level modules. Both should depend on abstractions (interfaces).**
2. **Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.**

In simple terms: Code should depend on **interfaces** or **abstract classes**, not on concrete classes. This makes the application more flexible and testable because implementations can be easily swapped out.

---

9. What is the difference between `==` and the `.Equals()` method?

These two are used for comparison, but their behavior varies based on the type being compared:

Comparison	Value Types (`int`, `struct`)	Reference Types (`class`, `string`)
== Operator	Compares the **actual values**.	**Default:** Compares **memory addresses** (reference equality). **Exception:** For `string` and other types that overload the operator, it compares values.
`.Equals()` Method	Compares the **actual values**.	**Default:** Compares **memory addresses** (reference equality). **Overridden:** If the class overrides this method (like `string` does), it compares the **actual values** (value equality).

Export to Sheets

Best Practice: When comparing object contents, always use the virtual **`.Equals()`** method (after checking for null) to respect custom type logic.

---

10. What is a Tuple?

A **Tuple** is a lightweight structure used to hold a **fixed number of elements of potentially different types** without having to create a custom class or struct.

- **Before C# 7:** The `System.Tuple` class was used, accessing members as `Item1`, `Item2`, etc. (less readable).
- **C# 7.0+ (ValueTuple):** Introduced **Value Tuples** (structs) which are more efficient and support **named fields**, greatly improving readability.

Example (Value Tuple):

C#

```
// Naming the elements directly
(string Name, int Age, decimal Salary) employee = ("Sue", 30, 75000m);

// Accessing the elements
Console.WriteLine(employee.Name);
```

Basics & Fundamental Concepts

1. **What is C#?**
   Answer: C# is a modern, object-oriented programming language developed by Microsoft as part of the .NET platform. It supports strong typing, garbage collection, and many language features such as generics, LINQ, async/await, etc.
2. **What is the Common Language Runtime (CLR)?**
   Answer: CLR is the runtime environment of the .NET framework. It manages memory, executes code, provides services like garbage collection, exception handling, security, and JIT compilation.
3. **What is the difference between .NET Framework, .NET Core, and .NET (5/6/7)?**
   Answer:
   - .NET Framework is the original Windows-only implementation.
   - .NET Core is a cross-platform, modular, open-source framework.
   - From .NET 5 onwards, Microsoft merged .NET Core into a unified ".NET" platform, with future versions being cross-platform and performance optimized.
4. **What are Value Types vs Reference Types in C#?**
   Answer:
   - Value types store data directly (e.g., `int`, `struct`). They usually live on the stack.
   - Reference types store references to their data (e.g., `class`, `string`, arrays) which live on the heap.
5. **What is boxing and unboxing?**
   Answer:
   - Boxing is converting a value type to `object` or to an interface it implements (i.e. wrapping it in a reference).
   - Unboxing is extracting the value type from the object.
     This process has overhead and can incur performance costs.
6. **What is the difference between == and `Equals()` in C#?**
   Answer:

- `==` is an operator; depending on the type, it may compare references or values.
- `Equals()` is a method defined on `System.Object` and overridden by types to define equality logic. For reference types, default `Equals` is reference equality unless overridden.

7. **What are `ref` and `out` keywords? How do they differ?**

   Answer:

   - `ref` requires that the variable be initialized before passing to the method. The method can read and modify it.
   - `out` does not require initialization before passing, but the method must assign it before returning. It's intended for multiple return values.

8. **What is `var` keyword in C#?**

   Answer: `var` enables implicit typing — the compiler infers the compile-time type from the initializer. It is not dynamic typing.

9. **What are nullable types?**

   Answer: Nullable types allow value types to hold null: `Nullable<T>` or the shorthand `T?`. For example, `int?` can hold a valid `int` or `null`.

10. **What is `null coalescing operator` (`??`) and `null conditional operator` (`?.`)?**

    Answer:

    - `a ?? b` returns `a` if `a` is not null; otherwise, returns `b`.
    - `a?.Member` calls `Member` only if `a` is not null, else returns null.

11. **What is the difference between `const` and `readonly`?**

    Answer:

    - `const` is a compile-time constant; it must be assigned at declaration and can't change.
    - `readonly` can be assigned at declaration or in constructor, so it can vary per instance, but cannot be changed afterwards.

12. **What are namespaces in C# and why are they used?**

    Answer: Namespaces organize types (classes, interfaces, enums, etc.) and avoid name collisions by grouping related types.

13. **What is a struct and how is it different from a class?**

    Answer:

- Structs are value types; classes are reference types.
- Structs cannot have a default (parameterless) constructor or inherit from another class (except implicit from `ValueType`).
- Struct instances are copied on assignment; with classes, reference is copied.

14. **Can structs have methods, fields, and interfaces?**
    Answer: Yes. Structs can have fields, methods, properties, and implement interfaces. But they have limitations compared to classes (no inheritance, no default constructor, etc.).
15. **What is a sealed class?**
    Answer: A class declared with `sealed` cannot be inherited. Useful to prevent extensions or for performance optimizations.
16. **What is a partial class?**
    Answer: Partial classes let you split the definition of a class across multiple files (using the `partial` keyword). At compile time, they are combined into one class.
17. **What is an interface? How is it different from an abstract class?**
    Answer:
    - An interface defines a contract (methods, properties) without implementation (though with default interface methods in newer C#, some implementation is allowed).
    - An abstract class can provide both abstract and concrete members.
    - A class can implement multiple interfaces, but can inherit only one base class (abstract or concrete).
18. **When would you prefer an abstract class over interface and vice versa?**
    Answer:
    - Use an interface when you want multiple inheritance of behavior contracts.
    - Use abstract class when you want to share common implementation or state among derived classes.
19. **What is method overloading vs method overriding?**
    Answer:
    - Overloading: same method name, different parameter signatures within the same class.

   - Overriding: derived class provides its own implementation of a base class virtual/abstract method using `override`.

20. **What is the `virtual` keyword?**
    Answer: It marks a method or property in a base class as overridable by derived classes. Without virtual, you can't override.
21. **What is `abstract` method?**
    Answer: An abstract method has no implementation in the base class; derived classes must override it. The class containing it must be declared `abstract`.
22. **What is the `sealed` keyword when used on a method?**
    Answer: It prevents further overriding in derived classes. It's used in combination with `override sealed`.
23. **What is an indexer?**
    Answer: An indexer allows instances of a class or struct to be indexed like arrays (using `this[...]`). It provides custom logic to retrieve or assign items by index.
24. **What are properties (get/set)?**
    Answer: Properties are members that provide access (read/write) to class data (fields) in a controlled way. They wrap methods `get` and `set`, often with logic, validation, or access control.
25. **What are auto-implemented properties?**
    Answer: Properties where backing fields are automatically generated by the compiler, e.g. `public int Age { get; set; }`, without needing explicit private field.

Memory, Garbage Collection & Performance

26. **What is garbage collection (GC) in .NET?**
    Answer: GC is an automatic memory management system that frees objects that are no longer reachable. It handles allocating and releasing memory, and avoids many memory leaks common in manual memory management.
27. **When does garbage collection occur?**
    Answer: GC runs under multiple conditions: when memory pressure is high, when

generation thresholds are reached, when `GC.Collect()` is called (rarely), or when the system is idle.

28. **What are GC generations?**

    Answer: .NET divides objects into generations (0, 1, 2). New objects start in gen 0; if they survive collections, they move to higher generations. This optimizes for short-lived objects.

29. **What is `Dispose()` vs `Finalize()` (destructor)?**

    Answer:

    - `Dispose()` (from `IDisposable`) is called explicitly to free unmanaged resources deterministically.
    - `Finalize()` is called by the garbage collector before an object is reclaimed, to clean up unmanaged resources if `Dispose()` wasn't called.

30. **What is the `using` statement (or `using` block)?**

    Answer: It ensures that `IDisposable.Dispose()` is called at the end of the block, even in exceptions, providing deterministic disposal of resources.

31. **What is weak reference?**

    Answer: `WeakReference` allows referencing an object without preventing it from being collected by GC. Useful for caches where you want objects to be collected if memory is needed.

32. **What is object finalization and the dispose pattern?**

    Answer: The dispose pattern is a best practice combining `Dispose()` and `Finalize()` (via a protected virtual `Dispose(bool disposing)` method) to properly manage both managed and unmanaged resources and avoid resource leaks.

33. **What is memory leak in .NET? How can it happen?**

    Answer: Although GC handles memory, leaks can still happen if objects are rooted inadvertently (e.g., static references, event subscriptions not removed), preventing GC from collecting them.

34. **What is difference between `throw;` and `throw ex;` in exception handling?**

    Answer:

    - `throw;` rethrows the original exception (preserving stack trace).
    - `throw ex;` creates a new exception, losing the original stack trace context.

35. **What is `GC.Collect()`? Should you call it?**

    Answer: It forces garbage collection explicitly. Usually, you should avoid calling it manually; trust the runtime to manage GC. Use only in very specific scenarios.

36. **What does the `Optimize` attribute or settings do?**

    Answer: Compiler optimizations affect how code is inlined, unrolled, or reordered for performance. In debug builds, optimizations are often disabled for better debugging.

37. **What is the Large Object Heap (LOH)?**

    Answer: Objects over ~85,000 bytes are allocated in LOH, which is collected less frequently (only in gen 2). Fragmentation of LOH may be a performance issue.

38. **What is pinned object in .NET?**

    Answer: Pinning prevents GC from moving the object in memory (used when passing memory to unmanaged code). But too many pinned objects degrade performance.

39. **What are finalizers in C#?**

    Answer: Finalizers (declared via destructor syntax `~ClassName()`) are methods called by GC before reclaiming object, used to perform cleanup if `Dispose()` wasn't called.

40. **What is the difference between stack and heap?**

    Answer:

    - Stack holds local variables and call frames; allocation is fast; lifetime is short.
    - Heap holds objects with dynamic lifetime; GC handles allocation/deallocation; access is more expensive.

Delegates, Events, Lambdas, LINQ, etc.

41. **What is a delegate?**

    Answer: A delegate is a type-safe function pointer. It encapsulates a method (or list of methods) that can be called later.

42. **What is a multicast delegate?**

    Answer: A delegate that holds references to multiple methods. Invoking it calls all the methods in its invocation list.

43. **What is an event?**

    Answer: An event is a wrapper around delegate(s), exposing subscription/unsubscription via `+=` and `-=`. It restricts direct invocation from outside the class.

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

    Answer: A delegate is a type; an event is a mechanism enabling safer use of delegates by exposing only subscription methods (you can't directly reset event from outside, etc.).

45. **What are lambda expressions?**

    Answer: Lambda syntax (`x => x * 2`) provides concise, inline anonymous functions. They are often used in LINQ or to instantiate delegates.

46. **What is Func<>, Action<>, Predicate<>?**

    Answer:

    - `Func<T1, T2, …, TResult>` is a delegate type returning `TResult`.
    - `Action<T1, …>` is a void-returning delegate.
    - `Predicate<T>` is a `Func<T, bool>`—used to check a condition.

47. **What is LINQ?**

    Answer: LINQ (Language Integrated Query) enables querying collections, databases, XML, etc. with a uniform syntax in C#.

48. **What are the difference between `IEnumerable<T>` and `IQueryable<T>`?**

    Answer:

    - `IEnumerable<T>` is for in-memory collections; executing LINQ queries is done in memory.
    - `IQueryable<T>` enables pushing query logic to data sources (like databases) for deferred execution and provider-based optimization.

49. **What is deferred execution in LINQ?**

    Answer: Queries aren't executed when defined but when iterated (e.g., via `foreach`, `ToList()`, etc.). This allows chaining and query re-evaluation on each iteration.

50. **What is immediate execution in LINQ?**

    Answer: Methods like `ToList()`, `Count()`, `First()` force immediate execution of the query.

51. **What is expression tree in LINQ?**

    Answer: An expression tree is a data structure representing code in a tree form (e.g., `x => x + 1`). Used by LINQ providers to parse and translate queries (e.g., to SQL).

52. **What is PLINQ?**

    Answer: Parallel LINQ (PLINQ) provides a parallelized version of LINQ for data parallelism, automatically partitioning workloads across threads.

53. **What is extension method?**

    Answer: An extension method lets you "add" methods to existing types without modifying them, by defining a static method in a static class with the `this` keyword in the first parameter.

54. **What is the difference between `Select` and `SelectMany` in LINQ?**

    Answer:

    - `Select` projects each element into a result.
    - `SelectMany` flattens nested collections: for each element, you return a collection and `SelectMany` flattens into a single sequence.

55. **What is `GroupBy` in LINQ?**

    Answer: It groups elements by a key and returns groups. You can further project group data.

Collections & Generics

56. **What are generics in C#? Why use them?**

    Answer: Generics allow you to define classes, interfaces, and methods with a placeholder for the type (e.g. `List<T>`). They provide type safety, performance (no boxing), and code reuse.

57. **What is generic constraint?**

    Answer: You can restrict generic type parameters via constraints like `where T : class`, `where T : new()`, or `where T : SomeInterface`.

58. **What's the difference between `List<T>`, `IList<T>`, `IEnumerable<T>`, `Collection<T>`?**

    Answer:

    - `IEnumerable<T>`: read-only iteration.
    - `IList<T>`: adds indexing, insert/remove.
    - `List<T>`: concrete implementation of `IList<T>`.
    - `Collection<T>`: a base class meant for subclassing and providing more control.

59. **What is `Dictionary<TKey, TValue>`?**

    Answer: A generic collection mapping keys to values. Offers fast lookup by key. Keys must be unique.

60. **What is `HashSet<T>`?**

    Answer: A collection of unique elements, uses hashing internally. Useful for set operations and membership checking.

61. **What is `SortedList<TKey, TValue>` VS `SortedDictionary<TKey, TValue>`?**

    Answer:

    - `SortedList` uses array internally, good for small data sets, lower memory overhead.
    - `SortedDictionary` uses tree (RB tree) behind the scenes, better performance for frequent insert/delete.

62. **What is `Queue<T>` and `Stack<T>`?**

    Answer:

    - `Queue<T>` is FIFO (first in first out).
    - `Stack<T>` is LIFO (last in first out).

63. **What is `LinkedList<T>`?**

    Answer: A doubly linked list implementation allowing fast insertion and removal at any position, but slower random access.

64. **What is `ObservableCollection<T>`?**

    Answer: A collection that notifies when items get added, removed, or the whole list changes. Useful in data binding scenarios.

65. **What is covariance and contravariance in generics / delegates?**

    Answer:

- Covariance (e.g., `out` keyword) allows a more derived return type.
- Contravariance (e.g., `in` keyword) allows accepting less derived parameter types.
  They help in making generic interfaces more flexible.

66. **What is `IComparer<T>` vs `IComparable<T>`?**

    Answer:

    - `IComparable<T>` is implemented by a class to compare itself to another instance (via `CompareTo`).
    - `IComparer<T>` is an external comparison logic (you can pass custom comparer to collections).

Exception Handling, Error Handling

67. **How is exception handling done in C#?**

    Answer: Using `try` / `catch` / `finally`. You may optionally `throw` exceptions, catch specific types, and use `finally` for cleanup code.

68. **What is a custom exception? How do you create one?**

    Answer: You derive from `Exception` (or a subclass) and add constructors. Use custom exceptions to represent domain-specific errors.

69. **What is the difference between `finally` block and `finalize` method?**

    Answer:

    - `finally` is part of `try/catch` blocks and always executes after them (even in exceptions).
    - `finalize` is the destructor method called by the GC just before object reclamation.

70. **What is `AggregateException`?**

    Answer: A special exception that wraps multiple exceptions, typically from parallel or asynchronous operations.

71. **What is exception filtering (with `when` keyword)?**

    Answer: You can add a `when (condition)` clause in a `catch` block to only catch exceptions if the condition is true.

72. **What is `StackOverflowException` vs `OutOfMemoryException`?**

    Answer:

    - `StackOverflowException` occurs when the call stack limit is exceeded (infinite recursion, etc.).
    - `OutOfMemoryException` occurs when the system can't allocate more memory for the process.

73. **What is best practice for exception handling?**

    Answer: Don't swallow exceptions silently, catch only expected exceptions, prefer specific exceptions over catch-all, use `using` or `Dispose` for resources, wrap only where necessary, add meaningful context.

74. **When should you rethrow an exception?**

    Answer: When you catch an exception to add context or logging but still want it propagated. Use `throw;` (not `throw ex;`) to preserve stack trace.

75. **What is `Exception.InnerException`?**

    Answer: When an exception wraps another (e.g., when catching and rethrowing), the inner exception carries the original error details.

Multithreading, Async & Concurrency

76. **What is threading in .NET?**

    Answer: The ability to run multiple sequences of instructions concurrently (in parallel) using `System.Threading.Thread`, `Task`, `async/await`, etc.

77. **What is `ThreadPool`?**

    Answer: A pool of worker threads managed by .NET runtime, used for executing short-lived tasks instead of creating threads each time.

78. **What is `Task` and `Task<T>`?**

    Answer: Represents asynchronous operations. `Task` for void (no return), `Task<T>` for operations returning `T`. It integrates well with `async/await`.

79. **What is async/await?**

    Answer: Language features that allow asynchronous, non-blocking programming that looks like synchronous code. `async` marks a method, `await` waits for `Task`.

80. **What is `ConfigureAwait(false)`?**

    Answer: In `await`, `ConfigureAwait(false)` instructs that the continuation after await doesn't need to resume on the original synchronization context (good for library code to avoid deadlocks in UI contexts).

81. **What is `CancellationToken`?**

    Answer: It provides cooperative cancellation between threads or tasks. Methods check the token's state and abort when requested.

82. **What is `lock` keyword?**

    Answer: Ensures mutual exclusion by acquiring a monitor on an object, so only one thread can access the critical section at a time.

83. **What is `Monitor` class?**

    Answer: Lower-level synchronization primitive used by `lock` (which is syntactic sugar). Provides `Enter`, `Exit`, `Wait`, `Pulse`, `PulseAll`.

84. **What is `Semaphore` / `SemaphoreSlim`?**

    Answer: Controls access to a limited number of resources by multiple threads. `SemaphoreSlim` is lighter and more efficient.

85. **What is `ReaderWriterLockSlim`?**

    Answer: A lock that allows multiple concurrent readers but exclusive writers. Better performance when read operations dominate.

86. **What is deadlock? How can it occur?**

    Answer: A situation where two or more threads wait forever for locks held by each other. It occurs when locks are acquired in inconsistent order, or nested locks.

87. **What is race condition?**

    Answer: When multiple threads access and modify shared data simultaneously without proper synchronization, leading to unpredictable results.

88. **What is `volatile` keyword?**

    Answer: Ensures that a field is read/written directly from memory (not cached in registers), providing visibility across threads. But it doesn't guarantee atomicity.

89. **What is `Interlocked` class?**
    Answer: Provides atomic operations on variables (increment, decrement, compare-exchange) without locks.
90. **What is `Concurrent` collections (e.g. `ConcurrentDictionary`, `BlockingCollection`)?**
    Answer: Thread-safe collections designed for concurrent access, reducing the need for manual locking.

Design Patterns, Architecture, Advanced Topics

91. **What is the Singleton pattern and how to implement it in C#?**
    Answer: Singleton ensures only one instance of a class exists. Implementation: private constructor, a static readonly instance, and possibly `Lazy<T>`, double-checked locking, etc.
92. **What is the Factory pattern?**
    Answer: Factory pattern abstracts the creation logic so that clients request objects from factory methods without specifying the concrete classes.
93. **What is the Repository pattern?**
    Answer: A pattern to manage data access logic and provide a collection-like interface to domain/business logic, abstracting persistence layer.
94. **What is Dependency Injection (DI)?**
    Answer: A pattern where dependencies (services) are injected into classes rather than created internally. Helps with decoupling, testing, and maintainability.
95. **What is Inversion of Control (IoC)?**
    Answer: A broader principle where control flow of a program is inverted: the framework calls into user code rather than the user code calling libraries. DI is one way to achieve IoC.
96. **What is the SOLID principle?**
    Answer: A set of five design principles:
    - **S**: Single Responsibility Principle
    - **O**: Open/Closed Principle

   - **L**: Liskov Substitution Principle
   - **I**: Interface Segregation Principle
   - **D**: Dependency Inversion Principle

97. **What is CQRS (Command Query Responsibility Segregation)?**
    Answer: A pattern where you separate read operations (queries) from write operations (commands), often with different models.
98. **What is the difference between layered architecture vs onion / clean architectures?**
    Answer:
    - Layered architecture organizes by technical layers (UI, business, data).
    - Onion / Clean architectures organize around the domain, dependencies flow inward, and infrastructure details are outer layers depending on abstractions.
99. **What is reflection in C#?**
    Answer: Reflection in `System.Reflection` allows inspecting metadata (types, methods, properties) at runtime and dynamically invoking methods or creating instances.
100. **What is dynamic type in C#?**
    Answer: Using the `dynamic` keyword, operations are resolved at runtime (not compile time). It allows more flexible behavior but sacrifices compile-time safety.