Pre-Joining Topics

Week 4: C# Fundamentals

Types

Abstract:

Data **abstraction** is the process of hiding certain details and showing only essential information to the user.
Abstraction can be achieved with either **abstract classes** or interfaces

The abstract keyword is used for classes and methods:

Abstract class: is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).

An abstract class can have both abstract and regular methods.

It is not possible to create an object of the Abstract class.

Abstract method: can only be used in an abstract class, and it does not have a body. The body is provided by the derived class (inherited from).

It must be **overridden** in a derived class.

When to Use Abstract Classes

Use abstract classes when:

- You want to **provide a base class** with some **shared functionality**.
- You need **common code** and **some enforced methods** to be implemented by subclasses.
- Example: Shape, Vehicle, Animal, etc.

Access Modifiers in Abstract Methods

- Abstract methods are always **public** or **protected** — never **private**, because they must be accessible to derived classes.
- You cannot declare an **abstract static** or **abstract constructor**.

Abstract class provides **partial abstraction**

Advantages of Abstraction

1. **Hides Complexity**
   Abstraction hides the complex implementation details and exposes only the necessary functionalities to the user.
   Example: You can start a car using car.Start() without knowing how the ignition system works internally.
2. **Increases Code Reusability**
   Common functionalities can be written once in an abstract class and reused by multiple derived classes.
3. **Improves Maintainability**
   Since the implementation details are hidden, changes made to the internal logic do not affect other parts of the code that depend on the abstraction.
4. **Encourages Modular Design**
   The code becomes more organized and easier to manage as it is divided into separate modules — abstract base classes and derived implementations.
5. **Supports Flexibility and Scalability**
   New subclasses or functionalities can be added without modifying the existing code, following the *Open/Closed Principle* of OOP.
6. **Enhances Security**
   Abstraction prevents external entities from accessing or modifying sensitive data or logic, protecting the system's internal structure.

Disadvantages of Abstraction

1. **Increased Complexity for Developers**
   Designing and implementing abstract classes or interfaces requires good understanding of object-oriented design and relationships between classes.
2. **Cannot Be Instantiated**
   Abstract classes cannot be instantiated directly; they must be inherited by concrete classes, which can add extra development steps.
3. **Overhead of Maintenance**
   When too many abstract layers are used, maintaining and understanding the flow of control can become difficult.
4. **Reduced Performance**
   Abstraction can introduce a small performance overhead due to additional method calls or inheritance layers.

5. **Not Suitable for Small Projects**
   In simple applications, using abstraction can make the code unnecessarily complicated and harder to follow.

Sealed Classes in C#

Definition

A **sealed class** in C# is a class that **cannot be inherited**.
It is declared using the sealed keyword.

It's basically the **opposite of an abstract class**:

- **Abstract class** → must be inherited.
- **Sealed class** → cannot be inherited.

Why Use a Sealed Class?

1. **To prevent inheritance**
   - You may want to **stop other classes from extending** a class to ensure your design remains fixed.
2. **To enhance performance**
   - The JIT (Just-In-Time) compiler can make certain optimizations because it knows no class will derive from it.
3. **To provide security and integrity**
   - You can use sealed classes to prevent **accidental overriding** or **tampering** with sensitive logic.

Sealed Methods

You can also **seal** a method in a derived class — but only if it was **overridden** from a base class.

This prevents **further overriding** in subclasses.

```
class Base
{
   public virtual void Show()
   {
```

```
        Console.WriteLine("Base Show");


    }

    class Derived : Base

    {

       public sealed override void Show()

       {

          Console.WriteLine("Derived Show");

       }

    }


    class SubDerived : Derived

    {

       // public override void Show()  Error - cannot override sealed method

    }
```

Advantages of Sealed Classes

1. **Prevents Inheritance**
   A sealed class cannot be inherited, which helps in maintaining the integrity of your code and prevents accidental or unauthorized modifications through subclassing.
2. **Improves Performance**
   Since sealed classes cannot be inherited, the JIT (Just-In-Time) compiler can optimize calls to their methods, resulting in slightly faster execution.
3. **Enhances Security and Stability**
   Sealed classes protect critical functionality by preventing overriding of methods, ensuring stable and predictable behavior.
4. **Simplifies Design**
   Restricting inheritance reduces unnecessary complexity in code and helps enforce clear design boundaries.
5. **Ideal for Utility and Helper Classes**
   Sealed classes are well-suited for static or utility-like classes that are not meant to be extended.
   Example: Built-in .NET classes like System.String, System.Math, and System.Console are sealed.

Disadvantages of Sealed Classes

1. **Reduces Flexibility**
   Developers cannot extend or modify the functionality of sealed classes through inheritance.
2. **Hinders Testing or Mocking**
   In unit testing frameworks, sealed classes are difficult to mock since they cannot be inherited.
3. **Code Duplication Risk**
   When inheritance is restricted, similar code may have to be rewritten in multiple classes, leading to duplication.
4. **Violates the Open/Closed Principle**
   The OOP principle of being "open for extension but closed for modification" is restricted, as sealed classes cannot be extended.
5. **Not Suitable for Reusable Base Classes**
   If your class is intended for reuse or extension by other developers, sealing it will prevent them from building on top of it.

Feature	Abstract Class	Sealed Class
Instantiation	Cannot be instantiated directly	Can be instantiated directly
Inheritance	Must be inherited by a derived class	Cannot be inherited
Purpose	To provide a base for extension and polymorphism	To prevent further extension and protect implementation
Methods	Can have abstract methods (no body) and non-abstract methods (with body)	All methods must be fully implemented; can seal overridden methods
Access Modifiers	Members can be public, protected, or private	Members can be public, protected, or private; inheritance blocked
Design Principle	Encourages extension and code reuse	Encourages restriction and stability
Use Case	Base classes, frameworks, shared functionality	Utility classes, final implementations, security-sensitive classes
Real-World Analogy	Blueprint of a vehicle (must be extended)	Final report in a bank (cannot be modified)

Interface in C#

An interface in C# is defined using the interface keyword. It serves as a blueprint that declares methods, properties, events or indexers without implementation. A class or struct that implements the interface must provide the actual implementation, either implicitly or explicitly.

There are some important points to remember, as mentioned below:

- Interfaces cannot have private members.
- By default, all the members of Interface are public and abstract.
- The interface will always be defined with the help of the keyword interface.
- Interfaces cannot contain fields because they represent an implementation of data.
- We can achieve multiple inheritance with the interface but not with the classes.
- We can implement multiple interfaces in a single class separated by commas.

Why Use Interfaces?

1. To achieve full abstraction.
2. To support multiple inheritance in C# (since C# doesn't allow multiple base classes).
3. To define contracts that must be implemented by derived classes.
4. To decouple code — allowing flexibility in replacing implementations.

Advantages of Interfaces

- **Loose coupling:** It is used to achieve loose coupling rather than concrete implementations, we can decouple classes and reduce interdependencies.
- **Abstraction:** Interface helps to achieve full abstraction.
- **Maintainability:** It helps to achieve component-based programming that can make code more maintainable.
- **Multiple Inheritance:** Interface used to achieve multiple inheritance and abstraction.
- **Define architecture:** Interfaces add a plug and play like architecture into applications.
- **Modularity:** Promote a cleaner and more modular design. By separating the definition of behavior from the implementation

Disadvantages:

- **Too many interfaces can complicate design:**
If a class implements multiple interfaces, it can become **difficult to manage and understand**, leading to complex and cluttered designs.

- **No constructors:**
Interfaces cannot have constructors, so **common initialization logic** must be repeated in each implementing class.

- **Breaking changes are risky:**
Adding a new member to an existing interface forces **all implementing classes to update**, which can **break existing code** if not handled carefully.

- **Performance overhead (slight):**
Calling a method via an interface reference has a small performance cost compared to calling a method directly on a class. While usually negligible, it can accumulate in performance-critical applications.

Feature	Interface	Abstract Class
Methods	Only method signatures (C# 8+ allows default implementations)	Can have both abstract and concrete methods
Fields	Cannot have fields	Can have fields
Multiple Inheritance	Supported (a class can implement multiple interfaces)	Not supported (a class can inherit only one class)
Access Modifiers	Members are public by default	Can have public, protected, private members
Purpose	Contract or blueprint	Base class for code reuse and common implementation

Coupling in Software Engineering

Definition:
Coupling refers to the **degree of dependency between different modules, classes, or components** in a software system.

- **Low coupling (loose coupling):** Modules are **independent** and have minimal knowledge of each other.
- **High coupling (tight coupling):** Modules are **heavily dependent** on each other's implementation details.

Why Coupling Matters

- Lower coupling → **easier to maintain, extend, and test**.
- Higher coupling → **harder to modify**, as changes in one module can break others.

Generics:

Generics allow you to **define classes, methods, interfaces, or delegates with placeholders for data types**.

- This means you can write **type-safe and reusable code** without committing to a specific data type.
- The actual data type is specified **when the class, method, or interface is instantiated or called**.

Key idea: Generics help **avoid code duplication** and **reduce runtime errors**.

Why Use Generics

1. **Type safety:** Catch type errors at compile time.
2. **Code reusability:** Same class/method works with multiple data types.
3. **Performance:** Avoids boxing/unboxing for value types.

Advantages of Generics

1. **Type Safety:** Errors caught at compile-time.
2. **Reusability:** Same code works with multiple data types.
3. **Performance:** No boxing/unboxing for value types.
4. **Maintainability:** Less code duplication → easier to maintain.
5. **Consistency:** Uniform API for different data types.

Disadvantages of Generics

1. **Complexity:** Can make code harder to read for beginners.
2. **Limited Reflection:** Some generic type information is not available at runtime.
3. **Runtime Restrictions:** Cannot create instances of type parameters unless constrained (new() constraint required).
4. **Code Bloat:** Each unique type used with a generic class may create separate code in some cases.

Constraints allow you to **restrict the types** that can be used with generics.

where T : class

where T : struct → value type

where T : class → reference type

where T : BaseClass → inherits a base class

where T : interfaceName → implements an interface

Collections in C#

Definition:
Collections in C# are **objects used to store, manage, and manipulate groups of related objects**. Unlike arrays, collections are **dynamic** (can grow or shrink at runtime) and provide **many built-in methods** for searching, sorting, and managing data.

Collections are part of the **System.Collections**, **System.Collections.Generic**, and **System.Collections.Concurrent** namespaces.

Types of Collections in C#

1. Non-Generic Collections (System.Collections)

- Store objects of type object (no type safety).
- Boxing/unboxing occurs for value types (performance overhead).
- Examples:
  - ArrayList → Dynamic array.
  - Hashtable → Key-value pairs.
  - Stack → Last-in-first-out (LIFO) collection.
  - Queue → First-in-first-out (FIFO) collection.

2. Generic Collections (System.Collections.Generic)

- Type-safe, no boxing/unboxing.
- Examples:
  - List<T> → Dynamic array.
  - Dictionary<TKey, TValue> → Key-value pairs.
  - Stack<T> → Generic LIFO.
  - Queue<T> → Generic FIFO.
  - HashSet<T> → Unique unordered collection.

3. Concurrent Collections (System.Collections.Concurrent)

- Thread-safe collections for **multi-threaded applications**.
- Examples:
  - ConcurrentDictionary<TKey, TValue>
  - ConcurrentQueue<T>
  - ConcurrentBag<T>

Namespace: System.Collections.Concurrent

Designed for **multi-threaded applications**.

Avoids manual locking (lock keyword) in most cases.

Useful when **multiple threads** read/write to a collection concurrently.

Techniques Used

a) Fine-Grained Locking

- Instead of locking the **whole collection**, only a **portion (bucket, segment, node)** is locked.
- Example: **ConcurrentDictionary<TKey, TValue>** uses multiple **buckets** internally.
- When adding/removing a key, only the **specific bucket** is locked → other threads can operate on different buckets concurrently.
- This is called **lock striping**.

Benefit: Much better concurrency than a single global lock.

b) Lock-Free / Atomic Operations

- Collections like **ConcurrentQueue<T>** and **ConcurrentBag<T>** often use **atomic instructions**:
  - Interlocked.CompareExchange()
  - Interlocked.Exchange()
  - CAS (Compare-And-Swap)
- These **hardware-level atomic operations** allow a thread to update a pointer or counter safely **without a lock**.

Example:

- In ConcurrentQueue<T>, each node has a **Next pointer**.
- When enqueuing, a thread **atomically updates the tail pointer** using CAS → no other thread can corrupt it.

c) Thread-Local Storage (for ConcurrentBag<T>)

- Each thread keeps its **own local list of items** (thread-local storage).
- Most operations happen on the **thread's own bag** → very fast, no locking.
- Only when threads steal items from each other (for load balancing) is a minimal lock used.

Advantages of Collections

1. **Dynamic size** → No need to know size in advance.
2. **Rich methods** → Add, Remove, Search, Sort, Reverse, etc.
3. **Type safety with generics** → Avoids runtime errors.
4. **Supports iteration** → Can use foreach or LINQ.
5. **Thread-safe options** → Concurrent collections for multi-threaded apps.

Disadvantages of Collections

1. **Performance overhead** → Slightly slower than arrays due to extra features.
2. **Memory usage** → May use more memory than fixed-size arrays.
3. **Non-generic collections lack type safety** → Can lead to runtime errors.
4. **Complexity** → More complex than simple arrays for beginners.

Collection	Add	Remove	Access / Find	Iterate
ArrayList	O(1) amortized	O(n)	O(1) by index, O(n) by value	O(n)
Hashtable	O(1) avg	O(1) avg	O(1) avg	O(n)
Stack	O(1)	O(1)	O(1) top	O(n)
Queue	O(1)	O(1)	O(1) front	O(n)
Dictionary<TKey,TValue>	O(1) avg	O(1) avg	O(1) avg	O(n)

Serialization:

Serialization is the process of converting an object in memory into a format that can be stored (file, database) or transmitted (network) and later reconstructed (deserialization).

- **Serialization formats:** JSON, XML, Binary, YAML, etc.
- **Use cases:** Saving application state, sending objects over API/network, caching, configuration files.

Advantages of Serialization

1. **Persistence of Object State**
   - Objects can be saved to files, databases, or streams and later restored.
   - Example: Saving user settings or application state.
2. **Data Exchange**
   - Serialized data can be sent over a network or between applications in a standard format (JSON/XML).
   - Example: Web APIs using JSON or SOAP services using XML.
3. **Caching**
   - Objects can be serialized and stored in cache to improve performance.
   - Example: Frequently accessed configuration objects.
4. **Cross-Language Compatibility**
   - JSON/XML can be read by different programming languages.
   - Example: A C# backend sending JSON to a JavaScript frontend.
5. **Supports Object Cloning**
   - Deep cloning can be achieved by serializing an object and deserializing it into a new object.

Disadvantages of Serialization

1. **Performance Overhead**
   - Serialization and deserialization consume CPU and memory, especially for large objects.
2. **Data Size**
   - Serialized formats like XML can be verbose, leading to larger storage or transmission size.
3. **Versioning Issues**
   - If the class structure changes (fields added/removed), deserialization may fail or require custom handling.
4. **Security Risks**
   - Deserializing data from untrusted sources can lead to security vulnerabilities (e.g., code injection or object spoofing).
5. **Limited Control over Format**
   - Without attributes or converters, serialized data might not match exact requirements (field names, nesting).
6. **Not Always Suitable for Complex Objects**
   - Objects with non-serializable members (like database connections, threads, file handles) cannot be serialized directly.

JSON Serialization

JSON Serialization converts objects into JSON (JavaScript Object Notation) format, which is lightweight and widely used for web APIs, configuration files, and data exchange between web clients and servers.

The JsonSerializer class in C# is used for JSON serialization and deserialization. It enables objects to be converted into JSON format and vice versa. This class provides methods to serialize objects into JSON strings and deserialize JSON strings into objects, making it suitable for web APIs, configuration files, and data interchange between different platforms.

C# provides multiple ways to work with JSON:

1. **System.Text.Json** (recommended for .NET Core/.NET 5+)
2. **Newtonsoft.Json (Json.NET)** (popular, feature-rich) -**Install:** Install-Package Newtonsoft.Json

Newtonsoft.Json is a **popular, high-performance JSON framework** for .NET.

It allows you to **serialize** (convert objects to JSON) and **deserialize** (convert JSON to objects) easily.

Features include:

- Serialize/Deserialize objects, lists, dictionaries
- LINQ to JSON (JObject, JArray)
- Custom converters
- Flexible formatting

Very widely used in **web applications, APIs, and data processing**.

Feature/Aspect	**System.Text.Json**	**Newtonsoft.Json (Json.NET)**
Performance	Faster, lower memory	Slower, slightly heavier
Built-in / External	Built-in (.NET Core 3+)	External NuGet package
Ease of Use	Minimalistic API	Very flexible and feature-rich
Async Support	Yes (SerializeAsync)	Limited (manual workarounds)
Polymorphic Serialization	Limited	Excellent
Handling Circular Refs	Not supported by default	Supported with settings
Dynamic JSON/LINQ	No	Yes (JObject, JArray)
Community/Support	Growing	Very mature, widely used
Best For	High-performance, simple models	Complex models, advanced scenarios

XML Serialization

XML Serialization converts objects into XML format, making it human-readable and interoperable. It's commonly used for configuration files, web services, and data interchange between different platforms.

The **XmlSerializer** class in C# is used for XML serialization and deserialization. It allows objects to be converted into XML format and vice versa. This class provides methods to serialize objects into XML documents and deserialize XML documents into objects, making it suitable for various data interchange scenarios.

Pros of XML:

- Supports schemas and validation (XSD)
- Rich metadata (attributes, elements)
- Widely used in enterprise systems

Cons of XML vs JSON:

- Verbose, larger size
- Slower to parse

Feature	JSON	XML
Syntax	Lightweight, simple	Verbose, tag-based
Readability	Easy	Moderate
Data Types	Strings, numbers, arrays, bool	Only strings, attributes/elements
Parsing Speed	Fast	Slower
Schema Support	Limited	Strong (XSD)
Typical Use Cases	APIs, config, data exchange	Config, enterprise systems

.NET Base Libraries (BCL)

Library / Namespace	Purpose / Functionality
System	Core types like Object, String, DateTime, Math, Guid, etc.
System.Collections	Standard collections like ArrayList, Hashtable (legacy)
System.Collections.Generic	Generic collections like List<T>, Dictionary<TKey,TValue>, Queue<T>, Stack<T>
System.IO	File and stream operations (File, FileStream, StreamReader, StreamWriter)
System.Text	String manipulation, encoding, regex (StringBuilder, Encoding, RegularExpressions)
System.Text.Json	JSON serialization/deserialization (in .NET Core/.NET 5+)
System.Xml	XML processing (XmlDocument, XmlReader, XmlWriter)
System.Net	Networking (HttpClient, WebClient, Sockets)
System.Data	Core database access types (DataSet, DataTable, DataRow)
System.Data.SqlClient	SQL Server database operations (ADO.NET)
System.Threading	Multithreading support (Thread, ThreadPool, Task)
System.Threading.Tasks	Task-based asynchronous programming (Task, Task<T>, Parallel)
System.Reflection	Metadata inspection, runtime type info (Type, PropertyInfo, MethodInfo)
System.ComponentModel	Component and property management (INotifyPropertyChanged)
System.Linq	Language Integrated Query (Enumerable, Queryable)
System.Diagnostics	Logging, tracing, performance counters (Debug, Trace, Stopwatch)

Library / Namespace	Purpose / Functionality
System.Security	Security and cryptography (SHA256, RSA, X509Certificate)
System.Runtime	Low-level runtime support, memory management, serialization support

Lambda Expression

A **lambda expression** is a **concise way to represent an anonymous method** using **=> (lambda operator)**.

Introduced in **C# 3.0**, commonly used in **LINQ**, **delegates**, **events**, and **functional programming** style.

Lambda Expression Syntax

a) Expression Lambda (Single expression)

(parameters) => expression

- Returns the result of the expression automatically.

Example:

```
Func<int, int> square = x => x * x;

Console.WriteLine(square(5)); // Output: 25
```

b) Statement Lambda (Multiple statements)

(parameters) => { statements; }

- Use {} for **multiple statements**.
- Must use return if returning a value.

Example:

```
Func<int, int, int> add = (a, b) =>

{

   int sum = a + b;

   return sum;

};
```

Console.WriteLine(add(10, 20)); // Output: 30

c) No parameter Lambda

() => Console.WriteLine("Hello World");

- Useful for **Action delegates** or events.

Example:

Action greet = () => Console.WriteLine("Hello Lambda");

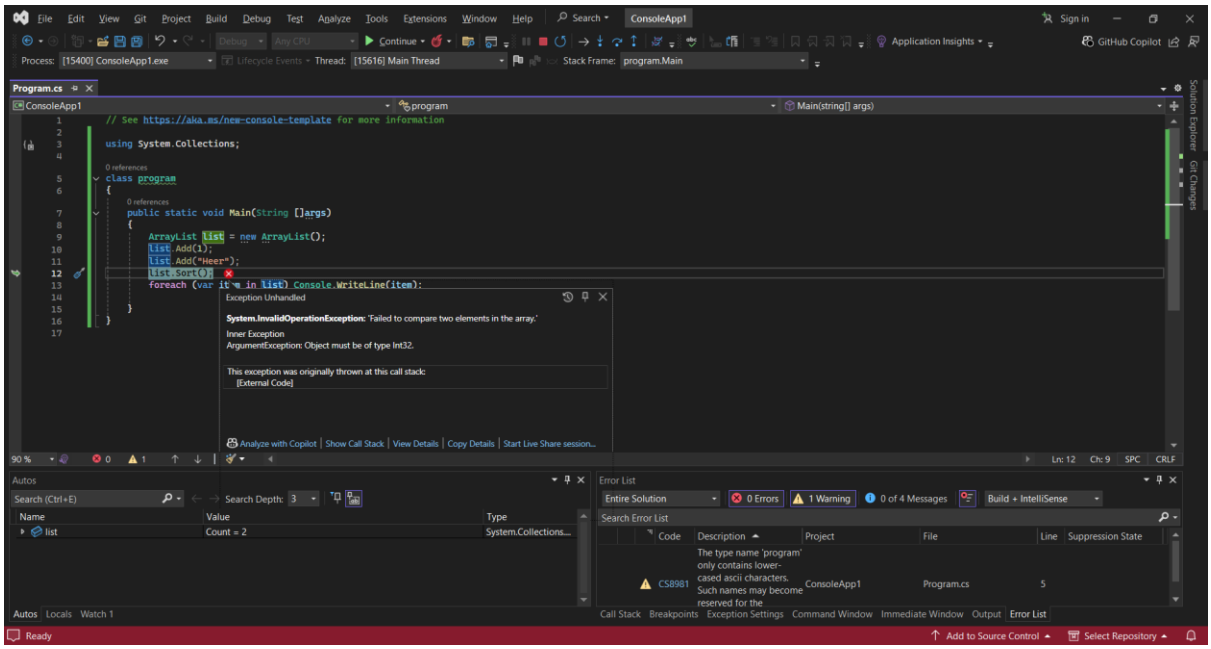
greet(); // Output: Hello Lambda

Advantages of Lambda Expressions

1. **Concise syntax** → replaces anonymous methods with fewer lines.
2. **Readable** → easier to understand small operations.
3. **Functional style** → works perfectly with LINQ.
4. **Reusable** → works with **Func, Action, Predicate** delegates.
5. **Powerful** → allows complex expressions in a simple way.

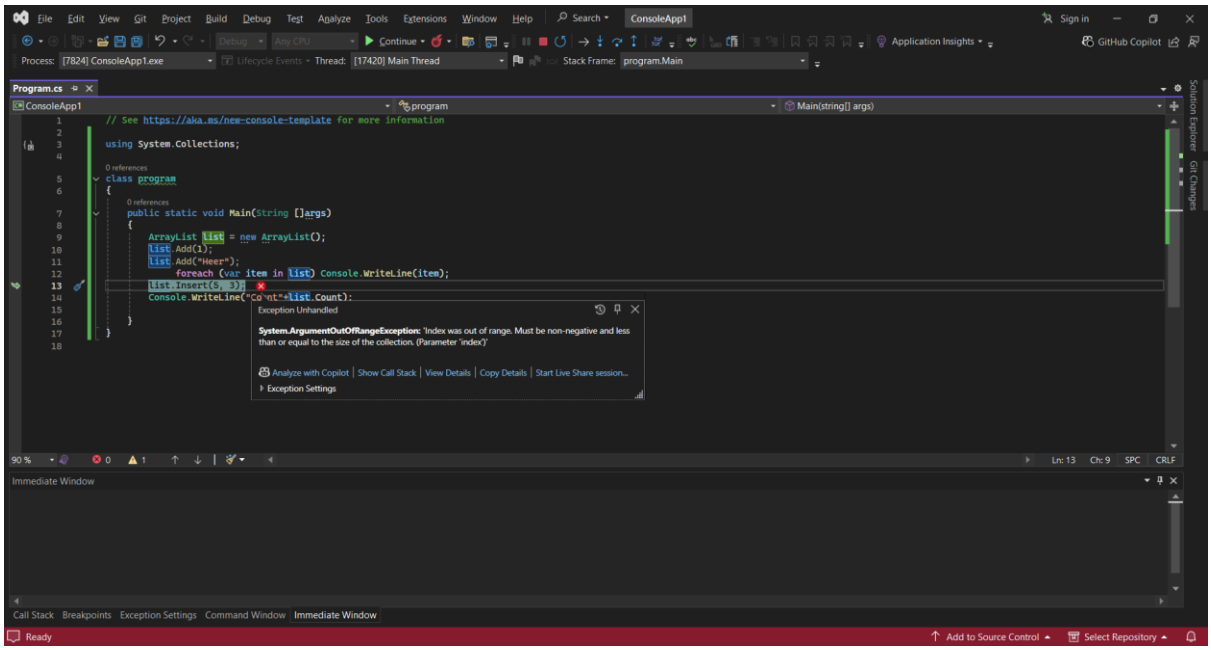
Questions During Reconcilation:

What is ArrayList has one int and other String then Sort?



Exception should be handled.

What if ArrayList has 2 item and new one is inserted at index 5 ?



Index value must be non negative and less than or equal to size of collection.

Technical examples when to use which type of collection ?

ArrayList

Example 1:

Loading CSV data where each row contains different data types temporarily before conversion.

Example 2:

Storing dynamic UI control values (TextBox, Label, ComboBox) before processing them.

Hashtable

Example 1:

Storing application configuration settings (before Dictionary was introduced).

Example 2:

Mapping user IDs (integer) to usernames (string) in a legacy system.

Stack (Non-Generic)

Example 1:

Undo/Redo functionality in a text editor.

Example 2:

Evaluating arithmetic expressions (postfix or prefix).

Queue (Non-Generic)

Example 1:
Printer job queue (first document sent, first printed).

Example 2:
Customer service ticket handling.

Generic Collections (System.Collections.Generic)

List<T>

Example 1:
Storing student objects for CRUD operations in a management system.

Example 2:
Managing a playlist in a music app.

Dictionary<TKey, TValue>

Example 1:
Caching user session data by user ID.

Example 2:
Mapping country codes to country names.

Stack<T>.

Example 1:
Browser navigation (Back/Forward buttons).

Example 2:
Recursive function call tracking (e.g., DFS in graphs).

Queue<T>

Example 1:
Multiplayer game matchmaking queue.

Example 2:
Task scheduling system (job execution in order).

HashSet<T>

Example 1:
Tracking unique visitors on a website.

Example 2:
Finding unique skills in resumes for filtering candidates.