**Part01**

**-Question: What is the difference between int.Parse and Convert.ToInt32 when**

**handling null inputs?**

1. **int.Parse:**
   * **Throws a ArgumentNullException if the input is null.**
   * **Example:**

**string input = null; int result = int.Parse(input); // Throws ArgumentNullException**

1. **Convert.ToInt32:**
   * **Returns 0 if the input is null.**
   * **Example:**

**string input = null; int result = Convert.ToInt32(input); // Returns 0**

**-Question: Why is TryParse recommended over Parse in user-facing applications?**

**TryParse is recommended over Parse in user-facing applications because:**

**Error Handling:**

**TryParse prevents exceptions by returning false for invalid input instead of throwing an exception.**

**Parse throws exceptions (FormatException or ArgumentNullException) for invalid or null inputs, requiring explicit handling.**

**Performance:**

**TryParse is faster because it avoids the overhead of exception handling.**

**User-Friendly:**

**TryParse simplifies validating user input without crashing the program, making it more robust for real-world applications.**

**Question: Explain the real purpose of the GetHashCode() method.**

**The GetHashCode() method provides a numerical value (hash code) that represents the object’s current state.**

**Purpose of GetHashCode():**

**-Efficient Lookups:**

**Used in collections like Dictionary, HashSet, and Hashtable to store and retrieve objects efficiently.**

**Helps quickly locate a bucket in hash-based data structures.**

**-Uniqueness (but not guaranteed):**

**Ensures that objects with different data usually have different hash codes.**

**Collisions (same hash code for different objects) are possible but rare.**

**-Equality Operations:**

**Works in conjunction with the Equals() method.**

**If two objects are considered equal by Equals(), their hash codes must also match.**

**Example:**

**string str1 = "Haneen";**

**string str2 = "Haneen";**

**Console.WriteLine(str1.GetHashCode() == str2.GetHashCode()); // True**

**Note: While GetHashCode() is essential for hash-based collections, it should not be used as a unique identifier.**

** Question: What is the significance of reference equality in .NET?**

**1-Shared Object Instances: Helps verify if two variables refer to the same object in memory.**

**object obj1 = new object();**

**object obj2 = obj1;**

**Console.WriteLine(Object.ReferenceEquals(obj1, obj2)); // True**

**2-Avoid Duplicate Work: Prevents unnecessary operations or duplication if the same object is being used.**

**3-Debugging Tool: Useful for identifying shared references and debugging memory-related issues.**

**Comparison Precision: Distinguishes between different objects with identical content (e.g., two strings with the same value but different instances).**

** Question: Why string is immutable in C# ?**

**1-Performance: Immutable strings allow for string interning, where identical string values are stored only once in memory, improving efficiency.**

**2-Security & Thread Safety: Immutability ensures that strings can't be modified by different threads, preventing data corruption.**

**3-Ease of Use: Immutable strings prevent accidental changes, making code more predictable and reducing bugs.**

**4-Implementation: Although backed by arrays of characters, any modification to a string creates a new object, leaving the original unchanged.**

** Question: How does StringBuilder address the inefficiencies of string concatenation?**

**StringBuilder addresses the inefficiencies of string concatenation by using a linked list internally instead of a static array. This allows it to efficiently grow and modify the string without creating new objects each time, unlike strings, which are immutable.**

* **Linked list: Instead of resizing a fixed-size array, StringBuilder uses a linked list of smaller chunks (buffers) to store the string, which enables efficient appending and modification.**
* **Performance: This reduces memory overhead and improves performance for repeated string concatenations.**

**Question: Why is StringBuilder faster for large-scale string modifications?**

**StringBuilder is faster for large-scale string modifications because:**

1. **Mutable String Storage: Unlike strings, which are immutable in C#, StringBuilder can modify the content of the string directly without creating new objects each time. This eliminates the overhead of creating a new string for each modification.**
2. **Efficient Memory Management: StringBuilder uses a buffer (often implemented as an array or linked list) to store the string data. It dynamically grows the buffer when needed, avoiding unnecessary memory allocations compared to concatenating strings repeatedly.**
3. **Reduced Copying: Each time you concatenate strings using the + operator, the entire string is copied into a new memory location. With StringBuilder, the string grows in place, reducing the number of copies required, especially when dealing with large strings.**

** Question: Which string formatting method is most used and why?**

**The most commonly used string formatting method in C# is String Interpolation (using the $ symbol) because:**

1. **Readability: String interpolation is more concise and easier to read compared to other methods like string.Format or concatenation. You directly embed the variables within the string, which makes the code look cleaner and more natural.**
2. **Performance: While the performance difference is generally negligible for small strings, string interpolation can be slightly faster than concatenation because it avoids creating multiple intermediate string objects.**
3. **Less Prone to Errors: In string interpolation, the placeholders are directly placed inside the string, reducing the chance of errors like mismatched placeholders, which can happen with string.Format.**
4. **Ease of Use: It allows you to mix expressions directly within the string (e.g., {num1 + num2}), making it more flexible and convenient.**

**Question: Explain how StringBuilder is designed to handle frequent modifications**

**compared to strings.**

**StringBuilder is more efficient than string for frequent modifications because:**

1. **Mutability: Unlike strings, which are immutable, StringBuilder allows in-place modifications without creating new objects each time.**
2. **Dynamic Buffer: It uses a dynamically resizing buffer, avoiding repeated memory allocations when appending or modifying content.**
3. **Performance: Frequent string concatenations or modifications (like in loops) are much faster with StringBuilder, as it doesn’t create new objects on every change.**
4. **Memory Efficiency: It minimizes memory overhead by expanding its buffer only when necessary, instead of creating a new string each time.**

**Part02**

**2- What’s Enum data type, when is it used? And name three common built\_in enums used**

**frequently?**

**Enum Data Type in C#**

**Definition: An enum (short for enumeration) is a distinct value type in C# that defines a set of named constants representing integral values. It allows developers to use meaningful names instead of numeric values, improving code readability and maintainability.**

**When is it Used? Enums are used when you need to define a set of related constants that represent underlying integral values, making your code easier to understand. Common use cases include:**

**1-Representing days of the week.**

**2-Defining status codes (success, failure).**

**3-Specifying options in a configuration or a setting (e.g., visibility levels).**

**Benefits of Using Enums:**

**1-Improves code readability by replacing magic numbers with descriptive names.**

**2-Reduces the chances of errors due to invalid values being used.**

**3-Makes code more maintainable, as the enum values are centralized.**

**Example of Enum Definition:**

**enum Day { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday }**

**Three Common Built-In Enums Used Frequently:**

**1-System.DayOfWeek**

**Represents the days of the week (Sunday, Monday, etc.), with underlying integer values starting from 0 (Sunday) to 6 (Saturday).**

**DayOfWeek day = DayOfWeek.Monday;**

**2-System.ConsoleColor**

**Represents colors used in the console (e.g., Red, Green, Blue, etc.). It helps change the text or background color in a console application.**

**Console.ForegroundColor = ConsoleColor.Green;**

**3-System.IO.FileMode**

**Specifies how the operating system should open a file (e.g., Open, Create, Append).**

**FileMode fileMode = FileMode.Open;**

**3- what are scenarios to use string Vs StringBuilder?**

**Scenarios to Use String vs StringBuilder**

**1. Use String When:**

* **Immutability is not a concern: If you do not need to modify the string multiple times, a regular string is fine because strings are immutable (unchangeable once created).**
* **Small or occasional string manipulations: For simple concatenations or modifications where performance is not critical. This is typical when dealing with a small number of strings or strings that are not frequently modified.**
* **Readability is more important: string operations are simpler and more readable when working with a small number of strings. Using the + operator for concatenation is intuitive and clean.**
* **Less frequent changes: If string concatenations or modifications are done just once or a few times, the performance difference between string and StringBuilder is negligible.**

**2. Use StringBuilder When:**

* **Frequent string modifications: If you need to perform multiple concatenations or modifications to a string, StringBuilder is more efficient. This is because StringBuilder works by modifying an internal buffer, whereas string creates a new object each time it is changed (due to its immutability).**
* **Large-scale string manipulation: If you are working with large strings or performing many operations on strings (e.g., building a string in a loop or constructing a large string from many pieces), StringBuilder is more efficient in terms of memory and performance.**
* **Performance-sensitive scenarios: In performance-critical code, especially when handling many string modifications or large amounts of data (e.g., reading and processing large text files, or constructing SQL queries dynamically), StringBuilder helps avoid the performance overhead that comes with string concatenation.**

**Part03 🡪Bonus**

**what meant by user defined constructor and its role in initialization?**

**A user-defined constructor is a special method in a class that is created by the programmer (not the default provided by C#).**

**It initializes object properties when the object is created.**

**Role in Initialization:**

**Sets initial values for object fields.**

**Ensures objects are ready for use immediately after creation.**

**Example:**

**class Person**

**{**

**public string Name;**

**public int Age;**

**// User-defined constructor**

**public Person(string name, int age)**

**{**

**Name = name;**

**Age = age;**

**}**

**}**

**Person person = new Person("Haneen", 20);**

**Console.WriteLine($"{person.Name}, {person.Age}"); // Output: Haneen, 20**

**-compare between Array and Linked List**

**Array:**

* **Fixed Size**: Requires predefined size.
* **Continuous Memory**: Stored in contiguous memory.
* **Access**: Fast (O(1)) via index.
* **Insertion/Deletion**: Slow (O(n)) due to shifting elements.
* **Efficient for**: Random access.

**Linked List:**

* **Dynamic Size**: Grows/shrinks as needed.
* **Scattered Memory**: Nodes linked via pointers.
* **Access**: Slower (O(n)), sequential traversal.
* **Insertion/Deletion**: Faster (O(1)) if pointer available.
* **Efficient for**: Frequent insertions/deletions.