Part 01

✅ Problem:

Write a program to:

1. Accept a string input from the user.
2. Convert it to an integer using both int.Parse and Convert.ToInt32.
3. Handle potential exceptions using a try-catch block.

```
using System;

class Program
{
  static void Main()
  {
    Console.Write("Enter a number: ");
    string input = Console.ReadLine();

    // Using int.Parse
    try
    {
      int parsedValue = int.Parse(input);
      Console.WriteLine("int.Parse result: " + parsedValue);
    }
    catch (Exception ex)
    {
      Console.WriteLine("int.Parse failed: " + ex.Message);
    }

    // Using Convert.ToInt32
    try
    {
            int convertedValue = Convert.ToInt32(input);
            Console.WriteLine("Convert.ToInt32 result: " + convertedValue);
        }
        catch (Exception ex)
        {
            Console.WriteLine("Convert.ToInt32 failed: " + ex.Message);
        }
    }
}
```

❓ Question:

What is the difference between int.Parse and Convert.ToInt32 when handling null inputs?

Answer:

- int.Parse(null) → Throws ArgumentNullException
- Convert.ToInt32(null) → Returns 0

So, Convert.ToInt32 is safer when dealing with potentially null values.

🔹 Next Section

✅ Problem:

Write a program that:

1. Prompts the user to input a number.
2. Uses int.TryParse to check if the input is valid.
3. If valid, print the number; otherwise, print an error message.

```
using System;
```

```
class Program
{
  static void Main()
  {
    Console.Write("Enter a number: ");
    string input = Console.ReadLine();

    if (int.TryParse(input, out int result))
    {
      Console.WriteLine("Valid number: " + result);
    }
    else
    {
      Console.WriteLine("Invalid input.");
    }
  }
}
```

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

Answer:

TryParse does not throw exceptions on invalid input—it simply returns false, making it:

- More efficient
- Safer in UI applications
- Better for handling unexpected or user-entered data

object variable and GetHashCode()

✅ Task:

1. Declare an object variable.
2. Assign it different types (int, string, double).
3. Print the GetHashCode() of each.

```
using System;

class Program
{
  static void Main()
  {
    object obj;

    obj = 42;
    Console.WriteLine($"int GetHashCode(): {obj.GetHashCode()}");

    obj = "Hello";
    Console.WriteLine($"string GetHashCode(): {obj.GetHashCode()}");

    obj = 3.14;
    Console.WriteLine($"double GetHashCode(): {obj.GetHashCode()}");
  }
}
```

Question: What is the real purpose of GetHashCode()?

Answer:

GetHashCode() returns an integer that represents the hash code of the object. It's mainly used in hashing algorithms and data structures like hash tables, dictionaries, and sets to quickly look up objects.

🔹 Problem 2: Reference behavior

✅ Task:

1. Create an object and assign it a value.
2. Create a second reference to the same object.

3. Modify the object using one reference.
4. Print it using the other reference.

```
using System;

class Sample
{
  public int Number;
}

class Program
{
  static void Main()
  {
    Sample obj1 = new Sample();
    obj1.Number = 10;

    Sample obj2 = obj1; // Reference to the same object

    obj1.Number = 99;

    Console.WriteLine("Value from obj2: " + obj2.Number); // Will print 99
  }
}
```

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

Answer:

Reference equality means two references point to the exact same object in memory. In .NET, you can check reference equality using Object.ReferenceEquals(obj1, obj2). This is important to know when determining if changes through one reference affect the other.

🔹 Problem 3: String immutability and GetHashCode()

✅ Task:

1. Declare a string and modify it by concatenating "Hi Willy".
2. Print GetHashCode() before and after modification.

```
using System;

class Program
{
  static void Main()
  {
    string original = "Hello";
    Console.WriteLine("Original GetHashCode(): " + original.GetHashCode());

    string modified = original + " Hi Willy";
    Console.WriteLine("Modified GetHashCode(): " + modified.GetHashCode());
  }
}
```

Why is string immutable in C#?

Answer:

Strings are immutable in C# for:

- Security: Safe for multi-threaded environments.
- Efficiency: Allows sharing and interning of strings.
- Predictability: Once created, the string value can't be changed, reducing side effects.

✅ Part 1: Using StringBuilder to Append Text

```
csharp
CopyEdit
using System;
using System.Text;

class Program
{
    static void Main()
    {
        StringBuilder sb = new StringBuilder("Hi Willy");
        Console.WriteLine("HashCode before append: " + sb.GetHashCode());

        sb.Append(", how are you?");
        Console.WriteLine("Modified text: " + sb.ToString());
        Console.WriteLine("HashCode after append: " + sb.GetHashCode());
    }
}
```

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

Answer: Strings in C# are immutable, meaning every time you change a string, a new object is created in memory. This leads to memory inefficiencies and slower performance when dealing with frequent changes. `StringBuilder` maintains a mutable buffer, modifying content without creating new string instances, improving memory usage and performance.

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

Answer: `StringBuilder` uses a dynamic buffer internally to store characters. Modifications like append, insert, replace, or remove work directly on this buffer without allocating new strings repeatedly, making it significantly faster for operations in loops or large data processing.

✅ Part 2: Program to Accept Inputs and Display Sum in 3 Ways

```
csharp
CopyEdit
using System;

class SumExample
{
    static void Main()
    {
        Console.Write("Enter first number: ");
        int num1 = int.Parse(Console.ReadLine());

        Console.Write("Enter second number: ");
        int num2 = int.Parse(Console.ReadLine());

        int sum = num1 + num2;

        // Method 1: String Concatenation
        Console.WriteLine("Sum is " + num1 + "+" + num2 + "=" + sum);

        // Method 2: Composite Formatting
        Console.WriteLine(string.Format("Sum is {0}+{1}={2}", num1, num2,
sum));

        // Method 3: String Interpolation
        Console.WriteLine($"Sum is {num1}+{num2}={sum}");
    }
}
```

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

Answer: String interpolation (`$`) is most widely used today because it is concise, readable, and easier to maintain. It directly embeds expressions within the string and avoids confusion from indexing used in `string.Format`.

✅ Part 3: StringBuilder Modification Program

```
csharp
CopyEdit
using System;
using System.Text;

class StringBuilderOperations
{
    static void Main()
    {
        StringBuilder sb = new StringBuilder("Hello C# Developer!");

        // Append
        sb.Append(" Welcome to StringBuilder.");
        Console.WriteLine("After Append: " + sb);

        // Replace
        sb.Replace("C#", "DotNet");
        Console.WriteLine("After Replace: " + sb);

        // Insert
        sb.Insert(6, "awesome ");
        Console.WriteLine("After Insert: " + sb);

        // Remove
        sb.Remove(6, 8); // removing "awesome "
        Console.WriteLine("After Remove: " + sb);
    }
}
```

🧠 Question: Explain how StringBuilder is designed to handle frequent modifications compared to strings.

Answer: `StringBuilder` allocates extra memory (buffer space) in anticipation of future changes. It avoids creating new instances on each modification. When the internal buffer is exhausted, it reallocates with a larger capacity. This approach enables frequent edits with lower memory overhead and faster performance than immutable strings.

2- What's Enum data type, when is it used? And name three common built_in enums used frequently? 3- what are scenarios to use string Vs StringBuilder?

What is an Enum data type, when is it used?

Definition:
`enum` (short for **enumeration**) is a **value type** in C# used to define a set of **named constants** that are logically related.

🧠 When is it used?

Enums are used when you have a variable that can take one out of a small set of predefined values — instead of using magic numbers or strings.

🔧 Example:

```
csharp
CopyEdit
enum DayOfWeek
{
    Sunday,
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
}

class Program
{
    static void Main()
    {
        DayOfWeek today = DayOfWeek.Wednesday;
        Console.WriteLine("Today is: " + today); // Output: Today is: 
Wednesday
    }
}
```

✅ Benefits:

- Improves **readability** and **maintainability**.
- Prevents invalid values.
- Easy to use with `switch` statements.

🔁 Three Common Built-in Enums:

1. `DayOfWeek` – represents days from Sunday to Saturday.
2. `ConsoleColor` – used to set foreground and background colors in the console.
3. `AttributeTargets` – defines application targets for custom attributes (used in reflection/attributes).

✅ 3. What are scenarios to use `string` vs `StringBuilder`?

Use `string` when:

- You are working with small or a fixed number of string operations.
- You are doing simple concatenation (1–2 times).
- Code readability and simplicity matter more than performance.
- Strings won't change after they are created.

```
csharp
CopyEdit
string fullName = "John" + " " + "Doe";
```

Use `StringBuilder` when:

- You perform **many or repeated modifications** (append, insert, remove).
- Working in **loops** that concatenate strings.
- Performance and memory optimization are important.

```
csharp
CopyEdit
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 1000; i++)
{
    sb.Append(i + " ");
}
```

⚖️ Summary Table:

Feature	`string`	`StringBuilder`
Immutable	✅ Yes	❌ No (mutable buffer)
Memory Efficient	❌ No (in loops)	✅ Yes (reuse buffer)
Best for	Few operations	Many/large operations
Performance	Slower (with changes)	Faster for repeated edits

What is a User-Defined Constructor and Its Role in Initialization?

🧱 Definition:

A **user-defined constructor** is a constructor that **you (the programmer)** create in a class to **control how objects are initialized** when they are created.

It sets **initial values** for the object's fields (or properties) when an object is instantiated.

🔧 Example:

```
csharp
CopyEdit
class Person
{
    public string Name;
    public int Age;

    // User-defined constructor
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }
}

class Program
{
    static void Main()
    {
        Person p = new Person("Alice", 25); // Constructor used here
        Console.WriteLine($"Name: {p.Name}, Age: {p.Age}");
    }
}
```

✅ Role of User-Defined Constructor:

- Allows **custom initialization** of objects.
- Supports **overloading** (you can have multiple constructors with different parameters).
- Helps enforce certain required fields when creating an object.

✅ 6. Compare: Array vs. Linked List

Feature	Array	Linked List
Memory Allocation	Contiguous block in memory	Nodes are scattered in memory, connected by links
Size	Fixed (must know size at creation)	Dynamic (can grow/shrink during runtime)
Access Speed	Fast: O(1) via index	Slow: O(n) – must traverse to find elements
Insertion/Deletion	Costly (needs shifting elements)	Easy (just update links)
Memory Usage	Efficient (no extra storage per item)	More memory (stores data **and** next node pointer)
Cache-Friendly	Yes (due to contiguous memory)	No
Usage Scenario	When size is known and fast access is needed	When frequent insertions/deletions are required

---

🔧 Example of Linked List in C#:

```
csharp
CopyEdit
using System;
using System.Collections.Generic;

class Program
{
    static void Main()
    {
        LinkedList<int> list = new LinkedList<int>();
        list.AddLast(10);
        list.AddLast(20);
        list.AddFirst(5);

        foreach (var item in list)
            Console.WriteLine(item);
    }
}
```