1: **What’s Enum data type?**

Enum (short for **enumeration**) is a **value type** in C# used to define a set of **named constants**.
Each name in an enum represents an underlying integer value.

- **Basic Syntax:**

  ```
  enum Days { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday }
  ```

  By default, Sunday = 0, Monday = 1, etc. You can also assign custom values:

  ```
  enum Status { Success = 1, Failed = 0, Pending = -1 }
  ```

❖ When is Enum used?

You use enum when you need to work with a fixed set of **related constants** that make the code **more readable**, **maintainable**, and **type-safe**.

- **Common use cases:**
  - Representing days of the week, user roles, error codes, status codes, etc.
  - Improving code clarity instead of using "magic numbers" or strings.
  - Used in switch statements for cleaner branching logic.

❖ Three common built-in enums used frequently in .NET:

1. **DayOfWeek**
   Represents the days of the week (Sunday to Saturday).

   ```
   DayOfWeek today = DateTime.Now.DayOfWeek;
   ```

2. **ConsoleColor**
   Represents colors you can use in the console.

   ```
   Console.ForegroundColor = ConsoleColor.Green;
   ```

3. **FileAccess** (from System.IO)
   Defines read/write permissions for file streams.

   ```
   FileStream fs = new FileStream("file.txt", FileMode.Open, FileAccess.Read);
   ```

2: what are scenarios to use string Vs StringBuilder?

➢ **Use string when:**

- You're working with **small or fixed text**.
- You're doing **minimal modifications** (e.g., a few concatenations).
- You care more about **readability** than performance.
- You're using **interpolation** or **simple formatting**.

➢ **Examples:**

```
string greeting = "Hello, " + name + "!";

string fullName = $"{firstName} {lastName}";
```

❖ **Why not for heavy modifications?**

In C#, strings are **immutable** — each change creates a **new string in memory**. If you modify it many times (like in a loop), it causes performance problems.

➢ **Use StringBuilder when:**

- You're doing **frequent or repetitive changes** (append, insert, remove, replace).
- You're building a string inside a **loop** or large process.
- You care about **performance and memory efficiency**.

➢ **Examples:**

```
StringBuilder sb = new StringBuilder();

for (int i = 0; i < 1000; i++)

{

   sb.Append("Line " + i + "\n");

}

string result = sb.ToString();
```

🔄 Quick Comparison:

Feature	string	StringBuilder
Mutable	❌ No (immutable)	✅ Yes
Performance	❌ Slower in loops	✅ Faster in loops
Readability	✅ Simple & clean	❌ Slightly more verbose
Use case	Light use	Heavy modification

3: what meant by user defined constructor and its role in initialization

❖ **What is a User-Defined Constructor?**

A **user-defined constructor** is a **special method in a class** that **you create** to **initialize objects** with specific values when they are created.

➢ In C#, a **constructor**:

- Has the **same name as the class**.
- **Does not have a return type** (not even void).
- Runs **automatically** when you create an object.

❖ **Role in Initialization**

- Helps **set initial values** for object fields/properties.
- Ensures the object **starts in a valid state**.
- Can allow **different ways** to create an object using **parameters**.

❖ **Key Points:**

1. A **user-defined constructor** lets you control **how objects are initialized**.
2. It can have **parameters** (called a **parameterized constructor**) or none (default constructor).
3. Helps **avoid uninitialized or invalid objects**.

4: compare between Array and Linked List

1. Memory Structure

- **Array:**
  - Stores elements **contiguously** in memory.
  - Size is **fixed** once created.
- **Linked List:**
  - Stores elements in **nodes**, each node contains **data + reference to next node**.
  - Memory is **scattered**, and size is **dynamic**.

---

2. Insertion & Deletion

- **Array:**
  - **Slow** for inserting/deleting in the **middle** because elements must be **shifted**.
  - **Fast** at the **end** if there is space (O(1) for last element in a dynamic array like List<T>).
- **Linked List:**
  - **Fast** insertion/deletion **anywhere** if you already have the reference to the node (just change pointers).
  - **No shifting** is required.

---

3. Accessing Elements

- **Array:**
  - **Direct access** using an index (O(1)).
  - Example: arr[3] is instant.
- **Linked List:**
  - **Sequential access** (O(n)), must traverse from the head to find a specific element.
  - Example: To reach the 5th element, start from the 1st.

---

4. Memory Usage

- **Array:**
  - Memory efficient because it only stores values.
  - But **wastes memory** if reserved size > used elements.
- **Linked List:**
  - Uses **extra memory** for pointers (Next/Previous references).
  - But **no memory waste** from unused capacity.

---

5. Practical Usage

- **Array:**
  - Best for **fixed-size collections** and **frequent random access**.
  - Example: Storing student grades, image pixels.
- **Linked List:**
  - Best for **dynamic collections** with **frequent insertion/deletion**.
  - Example: Undo operations, music playlist navigation.

- **Quick Comparison Table**

Feature	**Array**	**Linked List**
Memory	Contiguous	Non-contiguous (nodes)
Size	Fixed	Dynamic
Access	O(1)	O(n)
Insert/Delete	Slow (shift needed)	Fast (if node known)
Extra Memory	None	Pointer for each node