1. What is copy constructor?

A **copy constructor** is a special type of constructor in C# that creates a new object by copying the values of another existing object of the same type. It’s useful when you want to create a new instance that’s a copy of an already existing one.

public class Person

{

public string Name;

public int Age;

// Copy Constructor

public Person(Person other)

{

Name = other.Name;

Age = other.Age;

}

}

Here, the Person(Person other) constructor creates a new Person object by copying the Name and Age from another Person object. This way, you don’t need to manually copy each property.

1. [LinkedIn article about constructor and its types?](https://www.linkedin.com/posts/ahmed-b-ramzy_constructor-and-its-types-in-c-%D8%A7%D9%84%D9%80-constructor-activity-7272591741747159040-2l8h?utm_source=share&utm_medium=member_desktop)
2. What is Indexer, when used, as business mention cases u have to utilize it?

An **Indexer** is a special type of property that allows you to access elements in an object using array-like syntax. It is like creating an "index" for an object, so you can use square brackets [] to get or set values, just like with an array or a list.

public class EmployeeList

{

private string[] employees = new string[5];

// Indexer to access employees by their index

public string this[int index]

{

get { return employees[index]; }

set { employees[index] = value; }

}

}

the EmployeeList class has an indexer that lets you access employee names by their index in the employees array.

**When to Use an Indexer:**

1. **Collections or Lists**: When you want to provide a convenient way to access items in a custom collection or list-like object, an indexer lets users access elements just like they would with an array.
   * **Example**: A business system that manages a list of products and allows the user to get product details by index.
2. **Matrix-like Data**: For systems where data is stored in rows and columns (like a grid), an indexer can allow you to access elements with two indices, simplifying the code.
   * **Example**: A business application managing sales data where each row represents a transaction and each column represents a different attribute (e.g., date, amount, customer).
3. **Custom Data Structures**: When you need to create a custom data structure (e.g., a dictionary or a table) that behaves like an array or list, an indexer allows users to access its elements in a simple, intuitive way.
   * **Example**: A system managing employee records where you want to easily access employee information by employee ID or index.

**Why Use It?**

* **Simplifies Access**: It makes accessing elements in a custom object simpler and more natural, similar to accessing elements in an array or collection.
* **Encapsulation**: You can control how data is accessed or modified, providing an extra layer of abstraction.

In business scenarios, an indexer can help make your code cleaner and easier to use when dealing with collections of items like products, employees, transactions, or any list-like data structure.

1. Summarize keywords we have learnt last lecture?

**Access Modifiers (AM):**

* **internal** (default for classes and structs): Accessible within the same assembly.
* **public**: Accessible from anywhere.
* **private**: Accessible only within the same class or struct.
* **protected**: Accessible within the same class and derived classes.
* **internal protected**: Accessible within the same assembly and derived classes.

 **Types:**

* **struct**: Value type, stored on the stack, supports encapsulation but does not support inheritance. Supports some polymorphism.
* **class**: Reference type, stored on the heap, supports all four OOP pillars (encapsulation, inheritance, polymorphism, and abstraction).
* **interface**: A contract that defines method and property signatures but does not contain implementation. By default, members are public.

**Encapsulation:**

* Hides internal data (using private fields) and provides controlled access (through public methods or properties like getters/setters).

**Constructors (ctor):**

* Special methods used to initialize objects. They don’t return a value.
* Can be overloaded with different parameter lists.

**Properties:**

* Full properties: Include both getter and setter for more control over data.
* Automatic properties: The compiler generates the backing field and provides basic getter/setter functionality.

**Methods:**

* Functions inside classes or structs that define behaviors.
* Can be overridden (like ToString() to change the default string representation).

**Polymorphism:**

* Allows objects of different types to be treated as instances of the same type through inheritance and interfaces.

**Memory Allocation:**

* **Value types** (like structs) are stored on the stack, and **reference types** (like classes) are stored on the heap.
*  **Struct Example:** Employee struct with private fields, properties, constructors, and methods.
*  **Class Example:** Point class that represents a point in 2D space, with constructor, ToString method, and properties.