Question Answering Assignment

1. What is a copy constructor?

A **copy constructor** is a special constructor used to create a new object as a copy of an existing object of the same class. It takes an instance of the class as a parameter and initializes the new object with the same values as the existing object. This is useful when you want to create a duplicate of an object, either as a deep copy (copying all nested objects) or a shallow copy (copying only the top-level fields).

- **Purpose**: To initialize a new object with the state of an existing object.
- **Syntax**: It is a constructor that takes a parameter of the same class type.
- **Not Automatically Provided**: Unlike some languages (e.g., C++), C# does not automatically generate a copy constructor; you must explicitly define it.
- **Shallow vs. Deep Copy**:
    - **Shallow Copy**: Copies only the top-level fields (value types and references). Reference types still point to the same objects in memory.
    - **Deep Copy**: Recursively copies all nested objects, ensuring the new object is fully independent.

Example:

```
public class Car
{
   public string Model { get; set; }
   public int Year { get; set; }
   public Engine Engine { get; set; } // Reference type
```

```
    // Regular constructor
    public Car(string model, int year, Engine engine)
    {
        Model = model;
        Year = year;
        Engine = engine;
    }

    // Copy constructor (shallow copy)
    public Car(Car other)
    {
        Model = other.Model;
        Year = other.Year;
        Engine = other.Engine; // Shares same Engine object
    }
}

public class Engine
{
    public int Horsepower { get; set; }

    public Engine(int horsepower)
    {
        Horsepower = horsepower;
    }
}

class Program
{
    static void Main()
    {
        Engine engine = new Engine(200);
        Car car1 = new Car("Toyota", 2020, engine);
        Car car2 = new Car(car1); // Using copy constructor

        Console.WriteLine($"Car1: {car1.Model}, {car1.Year}, Engine
HP: {car1.Engine.Horsepower}");

        Console.WriteLine($"Car2: {car2.Model}, {car2.Year}, Engine
```

```
HP: {car2.Engine.Horsepower}");

        // Modify car2's engine to show shallow copy effect
        car2.Engine.Horsepower = 250;
        Console.WriteLine($"After change - Car1 Engine HP:
{car1.Engine.Horsepower}"); // Also changed

        Console.WriteLine($"After change - Car2 Engine HP:
{car2.Engine.Horsepower}");
    }
}
```

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 a class or structure to be accessed like an array, using index notation ([]). It provides a way to access data within an object using an index, similar to how you access elements in an array or collection. Indexers are defined using this keyword and can take one or more parameters of any type (not just integers).

Syntax of an Indexer

```
public <return_type> this[<parameter_type> index]
{
    get { /* Return value based on index */ }
    set { /* Set value based on index and value */ }
}
```

- **Return Type**: The type of data the indexer returns or accepts.
- **this**: Keyword used to define the indexer.
- **Parameters**: The index parameter(s) can be of any type (e.g., int, string, etc.).

- **get**: Retrieves the value at the specified index.
- **set**: *A*ssigns a value to the specified index, using the implicit value parameter.

Example of an Indexer

```
public class StringList
{
   private string[] items = new string[10];

   // Indexer
   public string this[int index]
   {
       get
       {
           if (index >= 0 && index < items.Length)
               return items[index];
           throw new IndexOutOfRangeException();
       }

       set
       {
           if (index >= 0 && index < items.Length)
               items[index] = value;
           else
               throw new IndexOutOfRangeException();
       }
   }
}

class Program
{
   static void Main()
   {
       StringList list = new StringList();
       list[0] = "Apple";  // Uses indexer to set value
       list[1] = "Banana";
```

```
        Console.WriteLine(list[0]); // Uses indexer to get value:
Apple
        Console.WriteLine(list[1]); // Banana
    }
}
```

When to Use Indexers

Indexers are used when you want to provide array-like access to an object's internal data, making the object behave like a collection. They are particularly useful when:

- The class represents a collection or container of items (e.g., a list, dictionary, or custom data structure).
- You want to simplify access to internal data without exposing the underlying storage mechanism.
- The data can be logically accessed using an index or key (e.g., integers, strings, or other types).

Business Use Cases for Indexers

1. **Custom Collection Classes**:
    - **Scenario**: A retail company manages a product catalog with thousands of items stored in a custom class. An indexer allows employees to access products by a product ID or SKU (string-based index).
2. **Data Analysis and Reporting**:
    - **Scenario**: A financial application stores time-series data (e.g., daily stock prices) in a custom class. An indexer allows analysts to access data by date or index for generating reports or charts.
3. **Inventory Management**:

    - **Scenario**: A warehouse management system tracks items in specific slots or bins. An indexer allows quick access to items by bin number or location code.

4. **Configuration or Settings Management**:
    - **Scenario**: A software application stores user settings or configuration values in a custom class. An indexer allows developers to access settings by a key (e.g., setting name).

5. **Matrix or Grid-Based Applications**:
    - **Scenario**: A gaming company develops a 2D game where the game board is a grid. An indexer allows developers to access grid cells using (row, column) coordinates.

3. Summarize keywords we have learnt last lecture

Concise summary of the keywords

- **private**: The private access modifier restricts access to a member (field, method, etc.) to only within the same class. It ensures encapsulation by hiding implementation details from external code, commonly used for sensitive data like internal state variables.
- **get**: The get accessor is part of a property in C# and defines the logic to retrieve the property's value. It allows controlled access to a private field, enabling read-only or computed properties, e.g., public int Age { get => age; }.
- **set**: The set accessor, also part of a property, defines the logic to assign a value to the property using the implicit value parameter. It enables controlled modification of a private field, e.g., public int Age { set => age = value; }.

- **value**: The value keyword is an implicit parameter used in a property's set accessor to represent the value being assigned. It allows the property to process or validate the incoming data before storing it, e.g., set { if (value > 0) age = value; }.
- **override**: The override keyword is used to provide a new implementation for a virtual or abstract method, property, or event inherited from a base class. It enables polymorphism, allowing derived classes to customize behavior, e.g., public override void Display() { ... }.
- **private protected**: The private protected access modifier restricts access to a member to the containing class or derived classes within the same assembly. It's a hybrid of private and protected, useful for tightly controlled inheritance scenarios.
- **protected**: The protected access modifier allows a member to be accessed within its class and by derived classes, even if they're in different assemblies. It supports inheritance while maintaining encapsulation, e.g., for base class fields that subclasses need.
- **internal**: The internal access modifier limits access to a member to within the same assembly. It's useful for exposing types or members to other classes in the same project but hiding them from external assemblies.
- **public**: The public access modifier allows unrestricted access to a member from any code, inside or outside the assembly. It's used for interfaces or methods intended for external use, e.g., API endpoints or public properties.
- **internal protected**: The internal protected access modifier allows access to a member within the same assembly or by derived classes in any assembly. It combines internal and protected, useful for framework-level code that needs broader inheritance access.