2100030414
P.Sweety

LAB SESSION-8 **PRE-LAB**

1.What are the Advantages of Delegates and Events?

Solution:

1. Decoupling of Components: Delegates and events allow components to be loosely coupled. Publishers can raise events without needing to know or directly reference specific subscribers. This promotes better separation of concerns and easier maintenance of the codebase.

2. Flexibility and Extensibility: Delegates and events provide a flexible way to extend functionality. New subscribers can be easily added to handle events without modifying existing code, enabling easy extensibility of applications.

3. Asynchronous Programming: Delegates and events can be used to implement asynchronous programming patterns. This allows applications to perform long-running tasks without blocking the main thread, improving responsiveness and user experience.

4. Encapsulation and Information Hiding: Events allow classes to encapsulate their internal state and implementation details while still allowing external components to be notified of important changes. This helps maintain a clean and understandable API.

5. Support for Observer Pattern:Delegates and events provide a way to implement the Observer pattern, where an object (the subject) maintains a list of dependents (observers) that are notified of state changes. This pattern is useful for implementing UI updates, eventdriven systems, and more. **IN-LAB:**

TASK1: Task **Delegates and events** refers to task **Collections**

To add the following new functionalities to the project created in task Collections:

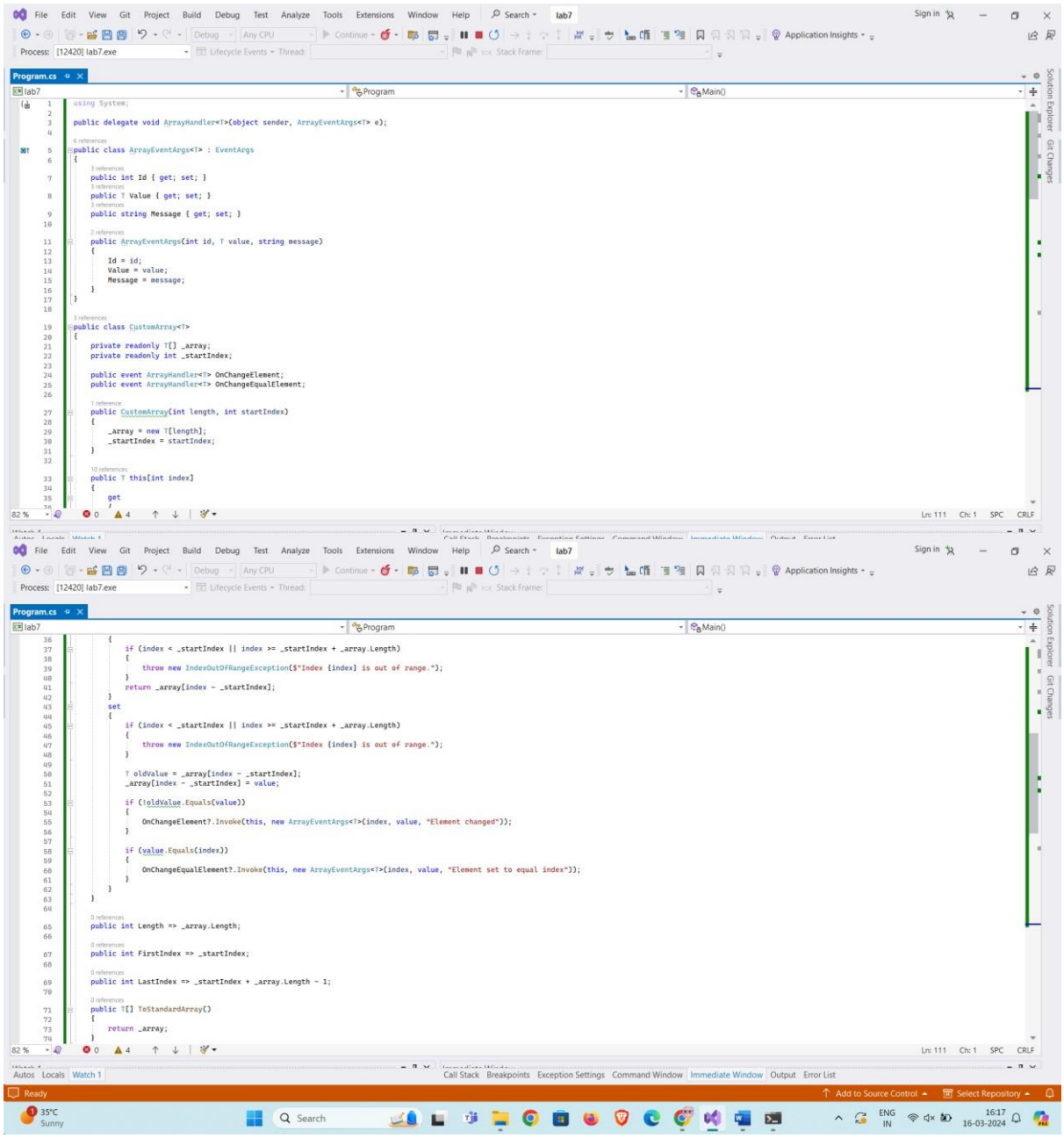
Include two events in the **CustomArray** type:

- The **OnChangeElement** event occurs when the indexer changes the element value (if the old and new element values match, the event is not raised)
- The **OnChangeEqualElement** event occurs if a value equal to the index of the changed element is written to the element (if the old and new values of the element match, the event is not raised)

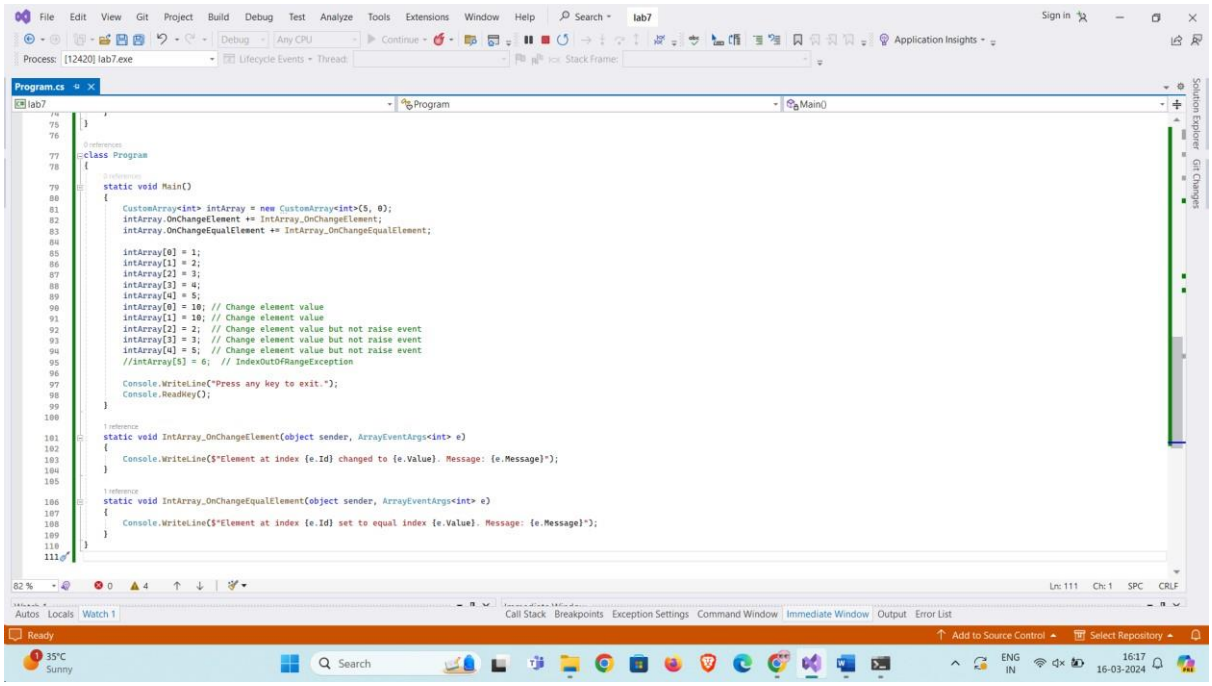
Use the **ArrayHandler** delegate to create the event.

The event handler takes two parameters: an object **sender** - a reference to the **CustomArray** instance that generated the event, and an event argument **ArrayEventArgs <T> e**. In the event argument, write in the **Id** field the index by which the user changes the element of the **CustomArray** array, in the **Value** field - the new value of the element by the Id index, in the **Message** field - an arbitrary string message.

Solution:

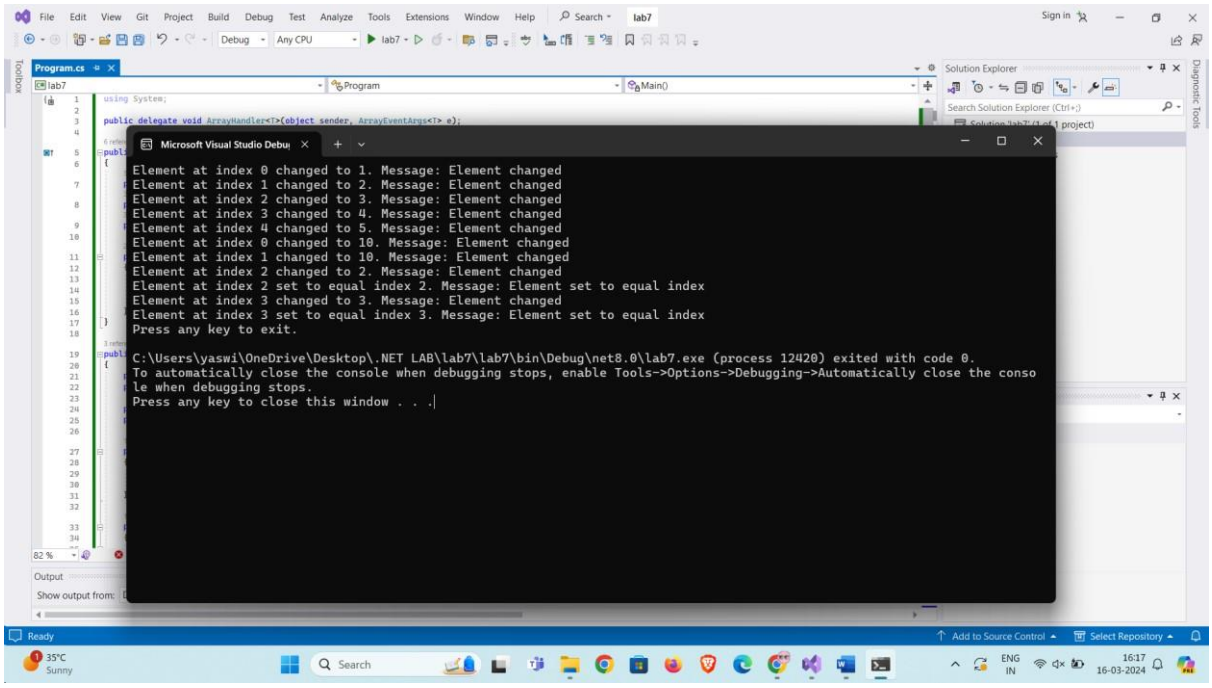


```
using System;

public delegate void ArrayHandler<T>(object sender, ArrayEventArgs<T> e);

public class ArrayEventArgs<T> : EventArgs
{
    public int Id { get; set; }
    public T Value { get; set; }
    public string Message { get; set; }

    public ArrayEventArgs(int id, T value, string message)
    {
        Id = id;
        Value = value;
        Message = message;
    }
}

public class CustomArray<T>
{
    private readonly T[] _array;
    private readonly int _startIndex;

    public event ArrayHandler<T> OnChangeElement;
    public event ArrayHandler<T> OnChangeEqualElement;

    public CustomArray(int length, int startIndex)
    {
        _array = new T[length];
        _startIndex = startIndex;
    }

    public T this[int index]
    {
        get
        {
            if (index < _startIndex || index >= _startIndex + _array.Length)
            {
                throw new IndexOutOfRangeException($"Index {index} is out of range.");
            }
            return _array[index - _startIndex];
        }
        set
        {
            if (index < _startIndex || index >= _startIndex + _array.Length)
            {
                throw new IndexOutOfRangeException($"Index {index} is out of range.");
            }

            T oldValue = _array[index - _startIndex];
            _array[index - _startIndex] = value;

            if (!oldValue.Equals(value))
            {
                OnChangeElement?.Invoke(this, new ArrayEventArgs<T>(index, value, "Element changed"));
            }

            if (value.Equals(index))
            {
                OnChangeEqualElement?.Invoke(this, new ArrayEventArgs<T>(index, value, "Element set to equal index"));
            }
        }
    }

    public int Length => _array.Length;

    public int FirstIndex => _startIndex;

    public int LastIndex => _startIndex + _array.Length - 1;

    public T[] ToStandardArray()
    {
        return _array;
    }
```



```
class Program
{
    static void Main()
    {
        CustomArray<int> intArray = new CustomArray<int>(5, 0);
        intArray.OnChangeElement += IntArray_OnChangeElement;
        intArray.OnChangeEqualElement += IntArray_OnChangeEqualElement;

        intArray[0] = 1;
        intArray[1] = 2;
        intArray[2] = 3;
        intArray[3] = 4;
        intArray[4] = 5;
        intArray[0] = 10; // Change element value
        intArray[1] = 10; // Change element value
        intArray[2] = 2;  // Change element value but not raise event
        intArray[3] = 3;  // Change element value but not raise event
        intArray[4] = 5;  // Change element value but not raise event
        //intArray[5] = 6;  // IndexOutOfRangeException

        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    static void IntArray_OnChangeElement(object sender, ArrayEventArgs<int> e)
    {
        Console.WriteLine($"Element at index {e.Id} changed to {e.Value}. Message: {e.Message}");
    }

    static void IntArray_OnChangeEqualElement(object sender, ArrayEventArgs<int> e)
    {
        Console.WriteLine($"Element at index {e.Id} set to equal index {e.Value}. Message: {e.Message}");
    }
}
```

OUTPUT:



```
Element at index 0 changed to 1. Message: Element changed
Element at index 1 changed to 2. Message: Element changed
Element at index 2 changed to 3. Message: Element changed
Element at index 3 changed to 4. Message: Element changed
Element at index 4 changed to 5. Message: Element changed
Element at index 0 changed to 10. Message: Element changed
Element at index 1 changed to 10. Message: Element changed
Element at index 2 changed to 2. Message: Element changed
Element at index 2 set to equal index 2. Message: Element set to equal index
Element at index 3 changed to 3. Message: Element changed
Element at index 3 set to equal index 3. Message: Element set to equal index
Press any key to exit.

C:\Users\yaswi\OneDrive\Desktop\.NET LAB\lab7\lab7\bin\Debug\net8.0\lab7.exe (process 12420) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

POST-LAB

1. Why and How the Delegates and Events are helpful to develop an Application?

Solution:

Delegates and events are helpful in application development for several reasons:

1. Decoupling:Delegates and events enable loose coupling between components. Publishers can raise events without needing to know or directly reference specific subscribers. This promotes better separation of concerns and easier maintenance of the codebase.

2. Flexibility and Extensibility:Delegates and events provide a flexible way to extend functionality. New subscribers can be easily added to handle events without modifying existing code, enabling easy extensibility of applications.

3. Asynchronous Programming:Delegates and events can be used to implement asynchronous programming patterns. This allows applications to perform long-running tasks without blocking the main thread, improving responsiveness and user experience.

4. Encapsulation and Information Hiding:Events allow classes to encapsulate their internal state and implementation details while still allowing external components to be notified of important changes. This helps maintain a clean and understandable API.

5.Support for Observer Pattern: Delegates and events provide a way to implement the Observer pattern, where an object (the subject) maintains a list of dependents (observers) that are notified of state changes. This pattern is useful for implementing UI updates, eventdriven systems, and more.