Міністерство освіти і науки України

Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського" Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

Звіт

з лабораторної роботи № 1 з дисципліни

«Сучасні технології розробки WEB-застосувань на платформі Microsoft.NET»

“**Узагальнені типи (Generic) з підтримкою подій. Колекції**”

Виконав(ла) *ІП-15 Костін В.А.* _____

(шифр, прізвище, ім'я, по батькові)

Перевірив *Бардін В.* _____

(прізвище, ім'я, по батькові)

Київ 2023

Лабораторна робота 1

Узагальнені типи (Generic) з підтримкою подій. Колекції

9	Динамічний масив з довільним діапазоном індексу	Див. List<T>	Збереження даних за допомогою вектору

Код виконання:

CustomArray.cs:

```
namespace CustomCollections
{
    public class CustomArray<T> : IList<T>
        where T : IComparable<T>, new()
    {
        #region PRIVATE FIELDS
        private readonly int _defaultCapacity = 4;

        private int _count = 0;
        private int _capacity;
        private T[] _items;
        #endregion

        #region CONSTRUCTORS
        public CustomArray()
        {
            _capacity = _defaultCapacity;
            _items = new T[_capacity];
        }

        public CustomArray(IEnumerable<T> items)
        {
            if (items is null)
            {
                throw new ArgumentNullException("Items cannot be null.");
            }

            _capacity = _defaultCapacity;
            _items = new T[_capacity];

            foreach (var item in items)
            {
                this.Add(item);
            }
        }

        public CustomArray(int capacity)
        {
            if (capacity < 0)
            {
                throw new ArgumentException("Capacity cannot be negative.");
            }
            else if (capacity > 0)
            {
                _capacity = _defaultCapacity = capacity;
                _items = new T[_capacity];
            }
            else
            {
```

```
                _capacity = capacity;
                _items = Array.Empty<T>();
            }
        }
        #endregion

        #region EVENTS
        public EventHandler<ArrayItemEventArgs<T>> ItemAdded;

        public EventHandler<ArrayItemEventArgs<T>> ItemRemoved;

        public EventHandler<ArrayEventArgs> ArrayCleared;

        public EventHandler<ArrayResizedEventArgs> ArrayResized;

        protected virtual void OnItemAdded(T item, int index)
        {
            if (ItemAdded != null)
            {
                ItemAdded(this, new ArrayItemEventArgs<T>(item, index,
ArrayAction.Add));
            }
        }

        protected virtual void OnItemRemoved(T item, int index)
        {
            if (ItemRemoved != null)
            {
                ItemRemoved(this, new ArrayItemEventArgs<T>(item, index,
ArrayAction.Remove));
            }
        }

        protected virtual void OnArrayCleared()
        {
            if (ArrayCleared != null)
            {
                ArrayCleared(this, new ArrayEventArgs(ArrayAction.Clear));
            }
        }

        protected virtual void OnArrayResized(int oldCapacity)
        {
            if (ArrayResized != null)
            {
                ArrayResized(this, new ArrayResizedEventArgs(oldCapacity,
_capacity));
            }
        }
        #endregion

        #region INTERFACE REALIZATION
        public int Count => _count;

        public bool IsReadOnly => false;

        public T this[int index] { get => _items[GetProperIndex(index)]; set =>
_items[GetProperIndex(index)] = value; }

        public void Add(T item)
        {
            if(item is null)
            {
                throw new ArgumentNullException("Item to add cannot be null.");
            }
```

```
            if (_count == _capacity)
            {
                Resize();
            }

            _items[_count++] = item;

            OnItemAdded(item, _count - 1);
        }

        public void Clear()
        {
            _count = 0;
            _capacity = _defaultCapacity;

            _items = new T[_capacity];

            OnArrayCleared();
        }

        public bool Contains(T item)
        {
            if (item is null)
            {
                throw new ArgumentNullException("Item to check for existance cannot
be null.");
            }

            for (int i = 0; i < _count; i++)
            {
                if (_items[i].CompareTo(item) == 0)
                    return true;
            }

            return false;
        }

        public void CopyTo(T[] array, int arrayIndex)
        {
            if (array is null)
            {
                throw new ArgumentNullException("Array cannot be null.");
            }

            if (array.Length - arrayIndex < _count)
            {
                throw new ArgumentOutOfRangeException("Number of elements to copy
cannot be placed into the destination array.");
            }

            Array.ConstrainedCopy(_items, 0, array, arrayIndex, _count);
        }

        public IEnumerator<T> GetEnumerator()
        {
            return new CustomEnumerator<T>(this);
        }

        public bool Remove(T item)
        {
            if (item is null)
            {
                throw new ArgumentNullException("Item to remove cannot be null.");
            }

            int index = IndexOf(item);
```

```
    if (index >= 0)
    {
        RemoveAt(index);

        return true;
    }

    return false;
}

IEnumerator IEnumerable.GetEnumerator()
{
    return GetEnumerator();
}

public int IndexOf(T item)
{
    if (item is null)
    {
        throw new ArgumentNullException("Item to search cannot be null.");
    }

    if (_count > 0)
    {
        for (int i = 0; i < _count; i++)
        {
            if (_items[i].CompareTo(item) == 0)
                return i;
        }
    }

    return -1;
}

public void Insert(int index, T item)
{
    if (item is null)
    {
        throw new ArgumentNullException("Item to insert cannot be null.");
    }

    if (_count == _capacity)
    {
        Resize();
    }

    int properIndex = GetProperIndex(index, toInsert: true);

    MovePartOfArray(properIndex);

    _items[properIndex] = item;

    _count++;

    OnItemAdded(item, properIndex);
}

public void RemoveAt(int index)
{
    int properIndex = GetProperIndex(index);

    var item = this[properIndex];

    MovePartOfArray(properIndex + 1, moveBack: true);
```

```
            _count--;

            OnItemRemoved(item, properIndex);
        }
        #endregion

        #region PRIVATE METHODS
        private int GetProperIndex(int index, bool toInsert = false)
        {
            if (_count == 0 && !toInsert)
            {
                throw new ArgumentOutOfRangeException("Cannot get or set element by
index: array is empty.");
            }

            int count = toInsert ? _count + 1 : _count;

            return index >= 0 ? index % count : (count + (index % count)) % count;
        }

        private void Resize()
        {
            int oldCapacity = _capacity;

            _capacity = oldCapacity is 0 ? _defaultCapacity : oldCapacity * 2;

            T[] tempArray = new T[_capacity];

            Array.Copy(_items, tempArray, _count);

            _items = tempArray;

            OnArrayResized(oldCapacity);
        }

        private void MovePartOfArray(int index, bool moveBack = false)
        {
            if((index == 0 && moveBack) || index >= _count)
            {
                return;
            }

            Array.Copy(_items, index, _items, moveBack ? index - 1 : index + 1,
_count - index);
        }
        #endregion
    }
}
```

Program.cs:

```
public static class Program
{
    public static void Foreach<T>(this IEnumerable<T> enumerable, Action<T> action)
    {
        foreach (var item in enumerable)
        {
            action(item);
        }
    }

    public static void Main()
    {
        //Initializing array and subscribing on it`s events
        CustomArray<int> array = new CustomArray<int>();

        array.ItemAdded += PrintArrayItemEventArgs!;
        array.ItemRemoved += PrintArrayItemEventArgs!;
        array.ArrayCleared += (sender, e) => Console.WriteLine($"--> Event invoked:
{e.Action} {e.ActionDateTime}");
        array.ArrayResized += (sender, e) =>
        Console.WriteLine($"--> Event invoked: {e.Action} {e.ActionDateTime} Old
capacity: {e.OldCapacity} New capacity: {e.NewCapacity}");

        //Adding new items to array
        array.Insert(0, 2);
        array.Insert(1, 3);
        array.Insert(2, 5);
        array.Insert(2, 4);
        array.Insert(0, 1);

        Console.WriteLine("\nArray after adding elements:");
        array.Foreach(item => Console.Write(item + " "));

        //Removing items from array
        Console.WriteLine("\n");

        array.Remove(3);

        array.RemoveAt(2);

        Console.WriteLine("\nArray after removing several elements:");
        array.Foreach(item => Console.Write(item + " "));

        //Clearing array
        Console.WriteLine("\n");

        array.Clear();
        Console.WriteLine($"Count: {array.Count}");
    }

    public static void PrintArrayItemEventArgs(object sender,
ArrayItemEventArgs<int> e)
    {
        Console.WriteLine($"--> Event invoked: {e.Action} {e.ActionDateTime} Item:
{e.Item} Index: {e.Index}");
    }
}
```

Результат роботи програми: