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

Лабораторна робота № 1

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

Викладач:

Бардін В.

Виконав:

студент 3 курсу

групи ІП-15 ФІОТ

Плугатирьов Д.В.

Київ – 2023

Тема: Узагальнені типи (Generic) з підтримкою подій. Колекції

Мета лабораторної роботи – навчитися проектувати та реалізовувати узагальнені типи, а також типи з підтримкою подій.

Завдання:

1. Розробити клас власної узагальненої колекції, використовуючи стандартні інтерфейси колекцій із бібліотек System.Collections та System.Collections.Generic. Стандартні колекції при розробці власної не застосовувати. Для колекції передбачити методи внесення даних будь-якого типу, видалення, пошуку та ін. (відповідно до типу колекції).
2. Додати до класу власної узагальненої колекції підтримку подій та обробку виключних ситуацій.
3. Опис класу колекції та всіх необхідних для роботи з колекцією типів зберегти у динамічній бібліотеці.
4. Створити консольний додаток, в якому продемонструвати використання розробленої власної колекції, підписку на події колекції.

Варіант	Опис узагальненої колекції	Функціонал	Реалізація
1	Стек	Див. Stack<T>	Збереження даних за допомогою динамічно зв’язаного списку

Програмний код

CustomLinkedListNode.cs

```
namespace CustomCollectionLib;

public sealed class CustomLinkedListNode<T>
{
    private T _value;

    public CustomLinkedListNode(CustomLinkedList<T> list, T value)
    {
```

```
        List = list ?? throw new ArgumentNullException(nameof(list));
        _value = value;
    }

    public CustomLinkedList<T>? List { get; private set; }

    public CustomLinkedListNode<T>? Next { get; set; }

    public CustomLinkedListNode<T>? Previous { get; set; }

    public T Value
    {
        get => _value;
        set => _value = value;
    }

    public ref T ValueRef => ref _value;

    public void Clear()
    {
        List = null;
        Next = null;
        Previous = null;
    }
}
```

CustomLinkedListEnumerator.cs

```
using System.Collections;

namespace CustomCollectionLib;

public class CustomLinkedListEnumerator<T> : IEnumerator<T>
{
    private readonly CustomLinkedList<T> _list;
    private CustomLinkedListNode<T>? _currentNode;
    private T? _currentValue;

    public T Current => _currentValue!;

    object? IEnumerator.Current => _currentValue;

    public CustomLinkedListEnumerator(CustomLinkedList<T> list)
    {
        _list = list;
        _currentNode = list.Head;

        if (list.Count != 0)
        {
            _currentValue = _currentNode!.Value;
        }
    }
```

```
    public bool MoveNext()
    {
        if (_currentNode is null)
        {
            return false;
        }

        _currentValue = _currentNode.Value;
        _currentNode = _currentNode.Next;

        if (_currentNode == _list.Head)
        {
            _currentNode = null;
        }

        return true;
    }

    public void Reset()
    {
        _currentValue = default;
        _currentNode = _list.Head;
    }

    public void Dispose()
    {
    }
}
```

CustomLinkedList.cs

```
using System.Collections;

namespace CustomCollectionLib;

public class CustomLinkedList<T> : ICollection<T>
{
    public CustomLinkedListNode<T>? Head { get; private set; }
    public int Count { get; private set; }

    public CustomLinkedList()
    {
    }

    public CustomLinkedList(ICollection<T> collection)
    {
        ArgumentNullException.ThrowIfNull(collection);

        foreach (var item in collection)
        {
            AddLast(item);
```

```
        }
    }

    public bool IsReadOnly => false;

    public CustomLinkedListNode<T>? First => Head;

    public CustomLinkedListNode<T>? Last => Head?.Previous;

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

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

    /// <exception cref="InvalidOperationException"></exception>
    private void ValidateNode(CustomLinkedListNode<T> node)
    {
        ArgumentNullException.ThrowIfNull(node);

        if (node.List != this)
        {
            throw new InvalidOperationException("The list node does not
belong to current list");
        }
    }

    void ICollection<T>.Add(T item)
    {
        AddLast(item);
    }

    public void AddAfter(CustomLinkedListNode<T> existingNode, T value)
    {
        ValidateNode(existingNode);

        var nodeToAdd = new CustomLinkedListNode<T>(this, value);
        InsertNodeBefore(existingNode.Next!, nodeToAdd);
    }

    public void AddBefore(CustomLinkedListNode<T> existingNode, T value)
    {
        var nodeToAdd = new CustomLinkedListNode<T>(this, value);
        InsertNodeBefore(existingNode, nodeToAdd);

        if (existingNode == Head)
        {
            Head = nodeToAdd;
        }
    }
```

```
    public CustomLinkedListNode<T> AddFirst(T value)
    {
        var result = new CustomLinkedListNode<T>(this, value);

        if (Head is null)
        {
            AddNodeToEmptyList(result);
        }
        else
        {
            InsertNodeBefore(Head, result);
            Head = result;
        }

        return result;
    }

    public CustomLinkedListNode<T> AddLast(T value)
    {
        var result = new CustomLinkedListNode<T>(this, value);

        if (Head is null)
        {
            AddNodeToEmptyList(result);
        }
        else
        {
            InsertNodeBefore(Head, result);
        }

        return result;
    }

    /// <exception cref="InvalidOperationException"></exception>
    private void AddNodeToEmptyList(CustomLinkedListNode<T> newNode)
    {
        if (Head is not null && Count != 0)
        {
            throw new InvalidOperationException("List must be empty when
this method is called!");
        }

        newNode.Next = newNode;
        newNode.Previous = newNode;
        Head = newNode;
        Count++;
    }

    private void InsertNodeBefore(CustomLinkedListNode<T> existingNode,
CustomLinkedListNode<T> newNode)
    {
        ValidateNode(existingNode);
```

```
        newNode.Next = existingNode;
        newNode.Previous = existingNode.Previous;
        existingNode.Previous!.Next = newNode;
        existingNode.Previous = newNode;

        Count++;
    }

    public void Clear()
    {
        var currentNode = Head;
        while (currentNode is not null)
        {
            var tempNode = currentNode;
            currentNode = currentNode.Next;
            tempNode.Clear();
        }

        Head = null;
        Count = 0;
    }

    public bool Contains(T value) => Find(value) is not null;

    public CustomLinkedListNode<T>? FindLast(T value)
    {
        if (Head is null) return null;

        var lastNode = Head.Previous;
        var currentNode = lastNode;

        if (currentNode is not null)
        {
            do
            {
                var comparer = EqualityComparer<T>.Default;
                if (comparer.Equals(currentNode!.Value, value))
                {
                    return currentNode;
                }

                currentNode = currentNode.Previous;
            } while (currentNode != Head);
        }
        else
        {
            do
            {
                if (currentNode!.Value is null)
                {
                    return currentNode;
                }

                currentNode = currentNode.Previous;
```

```
            } while (currentNode != lastNode);
        }

        return null;
    }

    public CustomLinkedListNode<T>? Find(T value)
    {
        var currentNode = Head;
        if (currentNode is null) return null;

        if (value is not null)
        {
            do
            {
                var comparer = EqualityComparer<T>.Default;
                if (comparer.Equals(currentNode!.Value, value))
                {
                    return currentNode;
                }
                currentNode = currentNode.Next;
            } while (currentNode != Head);
        }
        else
        {
            do
            {
                if (currentNode!.Value == null)
                {
                    return currentNode;
                }
                currentNode = currentNode.Next;
            } while (currentNode != Head);
        }

        return null;
    }

    /// <exception cref="ArgumentOutOfRangeException"></exception>
    /// <exception cref="ArgumentException"></exception>
    public void CopyTo(T[] array, int index)
    {
        ArgumentNullException.ThrowIfNull(array);

        if (index < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(index), index,
"Non-negative number required");
        }

        if (index > array.Length)
        {
            throw new ArgumentOutOfRangeException(nameof(index), index,
                "Must be less than or equal to the size of the
```

```
collection");
        }

        if (array.Length - index < Count)
        {
            throw new ArgumentException("Insufficient space in the target
location to copy the information");
        }

        var tempNode = Head;
        if (tempNode is null) return;

        do
        {
            array[index++] = tempNode!.Value;
            tempNode = tempNode.Next;
        } while (tempNode != Head);
    }

    public bool Remove(T value)
    {
        var nodeToRemove = Find(value);
        if (nodeToRemove is null) return false;

        RemoveNode(nodeToRemove);
        return true;
    }

    public void RemoveFirst()
    {
        if (Head is null) throw new InvalidOperationException("This method
shouldn't be called on empty list!");
        RemoveNode(Head);
    }

    public void RemoveLast()
    {
        if (Head is null) throw new InvalidOperationException("This method
shouldn't be called on empty list!");
        if (Head.Previous is null) throw new
InvalidOperationException("The node doesn't exist");
        RemoveNode(Head.Previous);
    }

    /// <exception cref="InvalidOperationException"></exception>
    private void RemoveNode(CustomLinkedListNode<T> nodeToRemove)
    {
        ValidateNode(nodeToRemove);

        if (Head is null)
        {
            throw new InvalidOperationException("This method shouldn't be
called on empty list!");
        }
```

```
        if (nodeToRemove.Next == nodeToRemove)
        {
            if (Head != nodeToRemove && Count != 1)
            {
                throw new InvalidOperationException("This should only be
true for a list with only one node");
            }

            Head = null;
        }
        else
        {
            nodeToRemove.Next!.Previous = nodeToRemove.Previous;
            nodeToRemove.Previous!.Next = nodeToRemove.Next;

            if (Head == nodeToRemove)
            {
                Head = nodeToRemove.Next;
            }
        }

        nodeToRemove.Clear();
        Count--;
    }
}
```

CustomStack.cs

```
using System.Collections;

namespace CustomCollectionLib;

public class CustomStack<T> : IEnumerable<T>
{
    private readonly CustomLinkedList<T> _list = new();

    public event EventHandler? StackCleared;
    public event EventHandler<T>? ItemPushed;
    public event EventHandler<T>? ItemPopped;

    protected virtual void OnItemPopped(T value)
    {
        ItemPopped?.Invoke(this, value);
    }

    protected virtual void OnStackCleared()
    {
        StackCleared?.Invoke(this, EventArgs.Empty);
    }

    protected virtual void OnItemPushed(T value)
```

```
    {
        ItemPushed?.Invoke(this, value);
    }

    public bool TryPeek(out T result)
    {
        if (_list.Count == 0)
        {
            result = default!;
            return false;
        }

        result = _list.Last!.Value;
        return true;
    }

    /// <exception cref="InvalidOperationException"></exception>
    public T Peek()
    {
        if (_list.Count == 0)
            throw new InvalidOperationException("Stack empty");
        return _list.Last!.Value;
    }

    public void CopyTo(T[] array, int arrayIndex)
    {
        _list.CopyTo(array, arrayIndex);
    }

    public bool Contains(T value)
    {
        return _list.Contains(value);
    }

    public void Clear()
    {
        _list.Clear();
        OnStackCleared();
    }

    public bool TryPop(out T result)
    {
        if (_list.Count == 0)
        {
            result = default!;
            return false;
        }

        result = _list.Last!.Value;
        _list.RemoveLast();
        return true;
    }

    /// <exception cref="InvalidOperationException"></exception>
```

```
    public T Pop()
    {
        if (_list.Count == 0)
            throw new InvalidOperationException("Stack empty");

        var elementToRemove = _list.Last!.Value;
        _list.RemoveLast();

        OnItemPopped(elementToRemove);
        return elementToRemove;
    }

    public void Push(T value)
    {
        _list.AddLast(value);
        OnItemPushed(value);
    }

    public int Count => _list.Count;

    public IEnumerator<T> GetEnumerator()
    {
        return _list.GetEnumerator();
    }

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

StackMessageService.cs

```
using CustomCollectionLib;

namespace CustomCollectionTestingApp;

public class StackMessageService<T> : IDisposable
{
    private readonly CustomStack<T> _stack;

    public StackMessageService(CustomStack<T> stack)
    {
        _stack = stack;
        stack.StackCleared += OnStackCleared;
        stack.ItemPushed += OnItemPushed;
        stack.ItemPopped += OnItemPopped;
    }

    private static void OnItemPopped(object? source, T value)
    {
        Console.WriteLine($"The item \"{value}\" was popped from the
```

```
stack!");
    }

    private static void OnItemPushed(object? source, T value)
    {
        Console.WriteLine($"The item \"{value}\" was pushed to the
stack!");
    }

    private static void OnStackCleared(object? source, EventArgs e)
    {
        Console.WriteLine("The stack was cleared!");
    }

    public void Dispose()
    {
        _stack.ItemPopped -= OnItemPopped;
        _stack.StackCleared -= OnStackCleared;
        _stack.ItemPushed -= OnItemPushed;
    }
}
```

Program.cs

```
using CustomCollectionLib;
using CustomCollectionTestingApp;

var stack = new CustomStack<string>();
using var messageService = new StackMessageService<string>(stack);

stack.Push("K");
stack.Push("P");
stack.Push("I");

stack.Pop();

Console.WriteLine("\nStack content:");
foreach (var item in stack)
{
    Console.WriteLine(item);
}

Console.WriteLine($"Stack count: {stack.Count}");
stack.Clear();
Console.WriteLine($"Stack count: {stack.Count}");
```

Результат виконання програми

```
The item "K" was pushed to the stack!
The item "P" was pushed to the stack!
The item "I" was pushed to the stack!
The item "I" was popped from the stack!

Stack content:
K
P
Stack count: 2
The stack was cleared!
Stack count: 0

Process finished with exit code 0.
```

Висновок

Під час виконання цієї лабораторної роботи мені вдалося покращити розуміння структури деяких колекцій, принципу роботи реалізацій інтерфейсів перелічування та реалізувати кілька подій для новоствореної колекції з подальшою їх перевіркою у консольному додатку. Мені вдалося стати більш впевненим у своїх навичках роботи зі структурами даних.