



Міністерство освіти і науки України КПІ ім. Ігоря Сікорського

Факультет Інформатики та Обчислювальної Техніки

ЗВІТ

Лабораторна робота №1 з дисципліни

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

Перевірив:

Викладач кафедри ІСТ

ФІОТ

Бардін В.

Виконав:

Хрисанфов Дмитро

гр. ІК-13

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.

6	Словник	Див. Dictionary<TKey, TValue>	Збереження даних за допомогою динамічно зв'язаного списку або вектору
---	---------	-------------------------------------	---

Код програми

```
using System;
using Lab1;

namespace Lab1
{
    public class Item<TKey, TValue>
    {
        public TKey Key { get; set; }
        public TValue Value { get; set; }

        public Item(TKey key, TValue value)
        {
            Key = key;
            Value = value;
        }

        public Item() {}

        public override int GetHashCode()
        {
            return Key.GetHashCode();
        }

        public override string ToString()
```

```

        {
            return Value.ToString();
        }

        public override bool Equals(object? obj)
        {
            return base.Equals(obj);
        }
    }
}

```

namespace Lab1

```

{
    public class MyDictionary<TKey, TValue> : IDictionary<TKey,
TValue>
    {
        private readonly int _size = 100;
        private readonly Item<TKey, TValue>[] _items;
        private readonly List<TKey> _keys = new List<TKey>();

        public event Action<TKey, TValue> ItemAdded;
        public event Action<TKey> ItemRemoved;
        public event Action DictionaryCleared;

        public void SubscribeToEvents()
        {
            ItemAdded += (key, value) => Console.WriteLine($"Element
with key {key} and value {value} added.");
            ItemRemoved += key => Console.WriteLine($"Element with
key {key} is removed.");
            DictionaryCleared += () => Console.WriteLine("Dictionary is
cleared.");
        }

        public MyDictionary()
        {
            _items = new Item<TKey, TValue>[_size];
            SubscribeToEvents();
        }
    }
}

```

```
public TValue this[TKey key]
{
    get => Search(key);
    set
    {
        Add(key, value);
        ItemAdded?.Invoke(key, value);
    }
}
```

```
public ICollection<TKey> Keys => _keys;
```

```
public ICollection<TValue> Values
{
    get
    {
        List<TValue> values = new List<TValue>();
        foreach (var item in _items)
        {
            if (item != null)
            {
                values.Add(item.Value);
            }
        }
        return values;
    }
}
```

```
public int Count => _keys.Count;
```

```
public bool IsReadOnly => false;
```

```
public TValue Search(TKey key)
{

```

```

var hash = GetHashCode(key);

if (_keys.Contains(key))
{
    if (_items[hash] == null)
    {
        foreach (var item in _items)
        {
            if (item != null && item.Key.Equals(key))
            {
                return item.Value;
            }
        }
    }
    else if (_items[hash].Key.Equals(key))
    {
        return _items[hash].Value;
    }
    else
    {
        var placed = false;
        for (var i = hash; i < _size; i++)
        {
            if (_items[i] == null)
            {
                break;
            }

            if (_items[i].Key.Equals(key))
            {
                return _items[i].Value;
            }
        }

        for (var i = 0; i < hash; i++)

```

```

        {
            if (_items[i] == null)
            {
                break;
            }

            if (_items[i].Key.Equals(key))
            {
                return _items[i].Value;
            }
        }
    }
}

```

```

        throw new KeyNotFoundException($"Key '{key}' not found in
the dictionary.");
    }

```

```

public void Add(TKey key, TValue value)
{
    var hash = GetHash(key);

    if (_keys.Contains(key))
    {
        throw new ArgumentException($"Key '{key}' already
exists in the dictionary.");
    }
    if (_items[hash] == null)
    {
        _keys.Add(key);
        _items[hash] = new Item<TKey, TValue> { Key = key,
Value = value };
        ItemAdded?.Invoke(key, value);
    }
}

```

```

else
{
    var placed = false;
    for (var i = hash; i < _size; i++)
    {
        if (_items[i] == null)
        {
            _keys.Add(key);
            _items[i] = new Item<TKey, TValue> { Key = key,
Value = value };
            placed = true;
            ItemAdded?.Invoke(key, value);
            break;
        }

        if (_items[i].Key.Equals(key))
        {
            return;
        }
    }

    if (!placed)
    {
        for (var i = 0; i < hash; i++)
        {
            if (_items[i] == null)
            {
                _keys.Add(key);
                _items[i] = new Item<TKey, TValue> { Key = key,
Value = value };
                placed = true;
                ItemAdded?.Invoke(key, value);
                break;
            }
        }
    }
}

```

```

        if (_items[i].Key.Equals(key))
        {
            return;
        }
    }
}

if (!placed)
{
    throw new Exception("Out of dictionary range");
}
}

}

public bool Remove(TKey key)
{
    var hash = GetHash(key);

    if (!_keys.Contains(key))
    {
        throw new InvalidOperationException($"$Key '{key}' not
found.");
    }
    if (_items[hash] == null)
    {
        for (var i = 0; i < _size; i++)
        {
            if (_items[i] != null && _items[i].Key.Equals(key))
            {
                _items[i] = null;
                _keys.Remove(key);
                ItemRemoved?.Invoke(key);
                return true;
            }
        }
    }
}

```



```
        throw new InvalidOperationException($"$Key '{key}' not  
found.");  
    }
```

```
    if (_items[hash].Key.Equals(key))  
    {  
        _items[hash] = null;  
        _keys.Remove(key);  
        ItemRemoved?.Invoke(key);  
        return true;  
    }
```

```
    else
```

```
    {  
        var placed = false;  
        for (var i = hash; i < _size; i++)  
        {  
            if (_items[i] == null)  
            {  
                throw new InvalidOperationException($"$Key '{key}'  
not found.");  
            }  
            if (_items[i].Key.Equals(key))  
            {  
                _items[i] = null;  
                _keys.Remove(key);  
                ItemRemoved?.Invoke(key);  
                return true;  
            }  
        }  
    }
```

```
    if (!placed)  
    {  
        for (var i = 0; i < hash; i++)  
        {  
            if (_items[i].Key.Equals(key))  
            {  
                _items[i] = null;  
                _keys.Remove(key);  
                ItemRemoved?.Invoke(key);  
                return true;  
            }  
        }  
    }
```

```
    if (!placed)  
    {  
        for (var i = 0; i < hash; i++)  
        {  
            if (_items[i].Key.Equals(key))  
            {  
                _items[i] = null;  
                _keys.Remove(key);  
                ItemRemoved?.Invoke(key);  
                return true;  
            }  
        }  
    }
```

```

        {
            if (_items[i] == null)
            {
                throw new InvalidOperationException($"$Key
'{key}' not found.");
            }

            if (_items[i].Key.Equals(key))
            {
                _items[i] = null;
                _keys.Remove(key);
                ItemRemoved?.Invoke(key);
                return true;
            }
        }
    }

    throw new InvalidOperationException($"$Key '{key}' not
found.");
}
}

```

```

public bool TryGetValue(TKey key, out TValue value)
{
    value = default(TValue);

    var hash = GetHash(key);

    if (!_keys.Contains(key))
    {
        throw new InvalidOperationException($"$Key '{key}' not
found.");
    }
    if (_items[hash] == null)

```

```

{
    foreach (var item in _items)
    {
        if (item != null && item.Key.Equals(key))
        {
            value = item.Value;
            return true;
        }
    }

    throw new InvalidOperationException($"$Key '{key}' not
found.");
}

if (_items[hash].Key.Equals(key))
{
    value = _items[hash].Value;
    return true;
}
else
{
    var placed = false;
    for (var i = hash; i < _size; i++)
    {
        if (_items[i] == null)
        {
            throw new InvalidOperationException($"$Key '{key}'
not found.");
        }

        if (_items[i].Key.Equals(key))
        {
            value = _items[i].Value;
            return true;
        }
    }
}

```

```

    }

    if (!placed)
    {
        for (var i = 0; i < hash; i++)
        {
            if (_items[i] == null)
            {
                throw new InvalidOperationException($"$Key
'{key}' not found.");
            }

            if (_items[i].Key.Equals(key))
            {
                value = _items[i].Value;
                return true;
            }
        }
    }

    throw new InvalidOperationException($"$Key '{key}' not
found.");
}

```

```

private int GetHashCode(TKey key)
{
    return key.GetHashCode() % _size;
}

```

```

public void Clear()
{
    var keysCopy = new List<TKey>(Keys);
}

```

```

        foreach (var key in keysCopy)
        {
            Remove(key);
        }

        DictionaryCleared?.Invoke();
    }

    public bool ContainsKey(TKey key)
    {
        foreach (var item in _items)
        {
            if (item != null && item.Key.Equals(key))
            {
                return true;
            }
        }

        throw new KeyNotFoundException($"$Key '{key}' not found.");
    }

    public bool ContainsValue(TValue value)
    {
        foreach (var item in _items)
        {
            if (item != null &&
                EqualityComparer<TValue>.Default.Equals(item.Value, value))
            {
                return true;
            }
        }

        throw new Exception($"Value '{value}' not found.");
    }

```

```
public void TryAdd(Item<TKey, TValue> item)
{
    if (_keys.Contains(item.Key))
    {
        throw new InvalidOperationException($"Key '{item.Key}'
already exists.");
    }
}
```

```
    Add(item.Key, item.Value);
}
```

```
public void Add(KeyValuePair<TKey, TValue> item)
{
    Add(item.Key, item.Value);
}
```

```
public bool Remove(KeyValuePair<TKey, TValue> item)
{
    if (_keys.Contains(item.Key))
    {
        Remove(item.Key);
        return true;
    }
    else
    {
        throw new Exception("Item not found.");
    }
}
```

```
public IEnumerator<KeyValuePair<TKey, TValue>>
GetEnumerator()
{
    foreach (var key in _keys)
    {
```

```

        var hash = GetHashCode(key);
        yield return new KeyValuePair<TKey, TValue>(key,
_items[hash].Value);
    }
}

```

```

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

```

```

public bool Contains(KeyValuePair<TKey, TValue> item)
{
    foreach (var key in _keys)
    {
        var hash = GetHashCode(key);
        var currentItem = _items[hash];

        if (currentItem != null &&
currentItem.Key.Equals(item.Key) &&
EqualityComparer<TValue>.Default.Equals(currentItem.Value,
item.Value))
        {
            return true;
        }
    }

    throw new Exception("Item not found.");
}

public void CopyTo(KeyValuePair<TKey, TValue>[] array, int
arrayIndex)
{
    if (array == null)
    {

```

```

        throw new ArgumentNullException(nameof(array));
    }

    if (arrayIndex < 0 || arrayIndex >= array.Length)
    {
        throw new
ArgumentOutOfRangeException(nameof(arrayIndex));
    }

    if (array.Length - arrayIndex < _keys.Count)
    {
        throw new ArgumentException("The destination array is
not large enough.");
    }

    int i = arrayIndex;
    foreach (var key in _keys)
    {
        var hash = GetHash(key);
        var currentItem = _items[hash];

        if (currentItem != null)
        {
            array[i++] = new KeyValuePair<TKey,
TValue>(currentItem.Key, currentItem.Value);
        }
    }
}
}
}

```

```

using Lab1;
using System;
using System.Collections.Generic;

```



```

namespace Lab1
{
    public class ConsoleApp
    {
        private readonly MyDictionary<int, string> _dict;

        public ConsoleApp()
        {
            _dict = new MyDictionary<int, string>();
        }

        public void Run()
        {
            while (true)
            {
                Console.Clear();
                Console.WriteLine("MyDictionary Console App");
                Console.WriteLine("=====");
                Console.WriteLine("Select an option:");
                Console.WriteLine("1. Add Key-Value Pair");
                Console.WriteLine("2. Search for a Key");
                Console.WriteLine("3. Remove a Key");
                Console.WriteLine("4. Display All Key-Value Pairs");
                Console.WriteLine("5. Clear Dictionary");
                Console.WriteLine("6. Exit");
                Console.Write("Enter your choice (1-6): ");

                if (int.TryParse(Console.ReadLine(), out int choice))
                {
                    switch (choice)
                    {
                        case 1:
                            Console.Clear();
                            Console.WriteLine("Add Key-Value Pair");
                            Console.WriteLine("=====");
                            Console.Write("Enter Key (int): ");
                            if (int.TryParse(Console.ReadLine(), out int key))
                            {
                                Console.Write("Enter Value (string): ");
                                string value = Console.ReadLine();
                                _dict.Add(key, value);
                                Console.WriteLine("\nKey-Value pair added
successfully.");
                            }
                            else
                            {
                                Console.WriteLine("\nInvalid input. Key must be an
integer.");
                            }
                            break;

                        case 2:
                            Console.Clear();
                            Console.WriteLine("Search for a Key");
                            Console.WriteLine("=====");
                            Console.Write("Enter Key to search for: ");
                            if (int.TryParse(Console.ReadLine(), out int searchKey))
                            {
                                string result = _dict.Search(searchKey);
                                Console.WriteLine($"Search result: {result}");
                            }
                            else
                            {
                                Console.WriteLine("\nInvalid input. Key must be an
integer.");
                            }
                            break;
                    }
                }
            }
        }
    }
}

```

```

        case 3:
            Console.Clear();
            Console.WriteLine("Remove a Key");
            Console.WriteLine("=====");
            Console.Write("Enter Key to remove: ");
            if (int.TryParse(Console.ReadLine(), out int removeKey))
            {
                bool removed = _dict.Remove(removeKey);
                if (removed)
                {
                    Console.WriteLine("\nKey-Value pair removed
successfully.");
                }
                else
                {
                    Console.WriteLine("\nKey not found in the
dictionary.");
                }
            }
            else
            {
                Console.WriteLine("\nInvalid input. Key must be an
integer.");
            }
            break;

        case 4:
            Console.Clear();
            Console.WriteLine("All Key-Value Pairs");
            Console.WriteLine("=====");
            foreach (var item in _dict)
            {
                Console.WriteLine($"Key: {item.Key}, Value:
{item.Value}");
            }
            break;

        case 5:
            Console.Clear();
            Console.WriteLine("Clear Dictionary");
            Console.WriteLine("=====");
            _dict.Clear();
            Console.WriteLine("\nDictionary cleared.");
            break;

        case 6:
            Console.Clear();
            Console.WriteLine("Exiting the application.");
            return;

        default:
            Console.Clear();
            Console.WriteLine("\nInvalid choice. Please select a
valid option.");
            break;
    }
}
else
{
    Console.Clear();
    Console.WriteLine("\nInvalid input. Please enter a number.");
}

Console.Write("\nPress Enter to continue...");
Console.ReadLine();
}
}
}

```

```

namespace Lab1
{
    public class Program
    {
        static void Main(string[] args)
        {
            var app = new ConsoleApp();
            app.Run();
        }
    }
}

```

Результати роботи програми

C:\Users\rofla\Desktop\3 курс\NET\Lab1\La

```

MyDictionary Console App
=====
Select an option:
1. Add Key-Value Pair
2. Search for a Key
3. Remove a Key
4. Display All Key-Value Pairs
5. Clear Dictionary
6. Exit
Enter your choice (1-6):

```

Методи Add/Remove/Search та підтримка подій:

<p>C:\Users\rofla\Desktop\3 курс\NET\Lab1\Lab1\bin\Debug</p> <pre> Add Key-Value Pair ===== Enter Key (int): 2 Enter Value (string): valueee Element with key 2 and value valueee added. Key-Value pair added successfully. Press Enter to continue... </pre>	<p>C:\Users\rofla\Desktop\3 курс\NET\Lab1\Lab</p> <pre> Search for a Key ===== Enter Key to search for: 2 Search result: valueee Press Enter to continue... </pre>
---	--

```
C:\Users\rofla\Desktop\3 курс\NET\Lab1\Lab1\
Remove a Key
=====
Enter Key to remove: 2
Element with key 2 is removed.

Key-Value pair removed successfully.

Press Enter to continue...
```

Метод Clear:

```
C:\Users\rofla\Desktop\3 курс\NET\Lab1\La
Clear Dictionary
=====
Element with key 3 is removed.
Dictionary is cleared.

Dictionary cleared.

Press Enter to continue...
```

Робота циклу foreach

```
C:\Users\rofla\Desktop\3 курс\NET
All Key-Value Pairs
=====
Key: 3, Value: kgrkerg

Press Enter to continue...
```

Обробка виключних ситуацій, в разі видалення елементу, ключа якого не існує

