

**МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ**  
**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ «КПІ»**



**Кафедра інформаційних систем та технологій**

**Лабораторна робота №1**

з дисципліни «Розробка програмного забезпечення на платформі .Net»

на тему:

«Узагальнені типи (Generic) з підтримкою подій. Колекції»

Викладачк:  
Бардін В.

Виконав:  
Студент групи ІС-11

Петраков Назар

**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:**

6	Словник	Див. <code>Dictionary&lt;TKey, TValue&gt;</code>	Збереження даних за допомогою динамічно зв'язаного списку або вектору
---	---------	---	---

Програмный код:

Entry.cs:

```
internal class Entry<TKey, TValue>
{
    public TKey key;
    public TValue? value;
    public int next;
    public uint hashCode;
}
```

MyDictionaryEventArgs.cs:

```
public class MyDictionaryEventArgs<TKey, TValue>: EventArgs
{
    public TKey? key;
    public TValue? value;

    public MyDictionaryEventArgs(TKey key, TValue value) : base()
    {
        this.key = key;
        this.value = value;
    }
}
```

MyDictionary.cs:

```
public class MyDictionary<TKey, TValue> : IDictionary<TKey, TValue>
{
    private Entry<TKey, TValue>[] _entries;
    private int[] _buckets;
    private int _capacity;
    private int _count = 0;

    public MyDictionary() : this(3) { }
    public MyDictionary(int capacity)
    {
        if(capacity < 0)
            throw new ArgumentOutOfRangeException(nameof(capacity));
        if(capacity is 0)
        {
            _buckets = Array.Empty<int>();
            _entries = Array.Empty<Entry<TKey, TValue>>();
        }
        _buckets = new int[capacity];
        _entries = new Entry<TKey, TValue>[capacity];
    }
}
```

```

        _capacity = capacity;
    }

    public event EventHandler<MyDictionaryEventArgs<TKey, TValue>>? AddedPair;
    public event EventHandler<MyDictionaryEventArgs<TKey, TValue>>? RemovedPair;
    public event EventHandler<MyDictionaryEventArgs<TKey, TValue>>? ChangedValue;
    public event EventHandler<EventArgs>? Cleared;

    public TValue this[TKey key]
    {
        get
        {
            Entry<TKey, TValue>? entry = FindEntryOfKey(key);

            if (entry == null)
                throw new KeyNotFoundException($"Key \"{key}\" is not found");
            return entry.value!;
        }
        set
        {
            if (key == null)
                throw new ArgumentNullException(nameof(key));

            Entry<TKey, TValue>? entry = FindEntryOfKey(key);

            if (entry == null)
                throw new KeyNotFoundException($"Key \"{key}\" is not found");
            entry.value = value;
            ChangedValue?.Invoke(this, new MyDictionaryEventArgs<TKey, TValue>(key,
value));
        }
    }

    public ICollection<TKey> Keys => GetCollection(entry => entry.key);

```

```

public ICollection<TValue> Values => GetCollection(entry => entry.value!);

public int Count => _count;

public bool IsReadOnly => false;

public void Add(TKey key, TValue value)
{
    if (key == null)
        throw new ArgumentNullException(nameof(key));
    if (ContainsKey(key))
        throw new ArgumentException("An item with the same key already exists in the
dictionary.");

    uint hashCode = (uint)key.GetHashCode();
    int bucketIndex = GetBucketIndex(hashCode);
    int entryIndex = -1;

    if (_count >= _capacity)
        Resize();

    for (int i = 0; i < _entries.Length; i++)
    {
        if (_entries[i] == null)
        {
            entryIndex = i;
            break;
        }
    }

    _entries[entryIndex] = new Entry<TKey, TValue>
    {
        key = key,

```

```

        value = value,
        hashCode = hashCode,
        next = _buckets[bucketIndex] - 1
    };
    _buckets[bucketIndex] = entryIndex + 1;
    _count++;
    AddedPair?.Invoke(this, new MyDictionaryEventArgs<TKey, TValue>(key, value));
}

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

public void Clear()
{
    _entries = new Entry<TKey, TValue>[_capacity];
    _buckets = new int[_capacity];
    _count = 0;
    Cleared?.Invoke(this, new EventArgs());
}

public bool Contains(KeyValuePair<TKey, TValue> item)
{
    if (item.Key is null)
        throw new ArgumentNullException(nameof(item.Key));

    Entry<TKey, TValue>? entry = FindEntryOfKey(item.Key);
    return entry is not null && EqualityComparer<TValue>.Default.Equals(item.Value,
entry.value);
}

public bool ContainsKey(TKey key)
{

```

```

        if (key is null)
            throw new ArgumentNullException(nameof(key));

        Entry<TKey, TValue>? entry = FindEntryOfKey(key);
        return entry is not null;
    }

    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 (_count > array.Length - arrayIndex)
            throw new ArgumentException("The destination array is not large enough to copy all
the elements.");

        foreach (var entry in _entries)
        {
            if (entry != null)
                array[arrayIndex++] = new KeyValuePair<TKey, TValue>(entry.key, entry.value!);
        }
    }

    public bool Remove(TKey key)
    {
        if (key == null)
            throw new ArgumentNullException(nameof(key));

        Entry<TKey, TValue>? entryToRemove = FindEntryOfKey(key);

        if (entryToRemove == null)
            throw new KeyNotFoundException($"Key \"{key}\" is not found");
    }

```

```

        for (int i = 0; i < _entries.Length; i++)
        {
            if (_entries[i] == entryToRemove)
            {
                _entries[i] = default!;
                _count--;

                RemovedPair?.Invoke(this, new MyDictionaryEventArgs<TKey,
TValue>(entryToRemove.key, entryToRemove.value!));

                return true;
            }
        }
        return false;
    }

    public bool Remove(KeyValuePair<TKey, TValue> item)
    {
        if (item.Key == null)
            throw new ArgumentNullException(nameof(item.Key));

        TKey key = item.Key;

        if (ContainsKey(key) && TryGetValue(key, out TValue? value) &&
EqualityComparer<TValue>.Default.Equals(value, item.Value))
        {
            Remove(key);

            return true;
        }

        return false;
    }

    public bool TryGetValue(TKey key, [MaybeNullWhen(false)] out TValue value)
    {
        if (key is null)

```



```
throw new ArgumentNullException(nameof(key));
```

```
Entry<TKey, TValue>? entry = FindEntryOfKey(key);
```

```
if (entry is null)
```

```
{
```

```
    value = default;
```

```
    return false;
```

```
}
```

```
value = entry.value!;
```

```
return true;
```

```
}
```

```
public IEnumerator<KeyValuePair<TKey, TValue>> GetEnumerator()
```

```
{
```

```
    return new MyEnumerator(this);
```

```
}
```

```
IEnumerator IEnumerable.GetEnumerator()
```

```
{
```

```
    return GetEnumerator();
```

```
}
```

```
public void Print()
```

```
{
```

```
    if (_count is 0)
```

```
        Console.WriteLine("No pairs in dictionary");
```

```
    else
```

```
{
```

```
    Console.WriteLine("-----\nYour dictionary:");
```

```
    foreach (var kvp in this)
```

```
{
```

```
        Console.WriteLine($"|key: {kvp.Key}|\t|value: {kvp.Value}|");
```

```
}
```

```

        Console.WriteLine("-----");
    }
}

private void Resize()
{
    int newCapacity = GetNextPrime(_capacity);
    var newEntries = new Entry<TKey, TValue>[newCapacity];
    _buckets = new int[newCapacity];

    Array.Copy(_entries, newEntries, _count);

    for (int i = 0; i < newCapacity; i++)
    {
        if (newEntries[i] is null)
            break;
        NewBucketIndexes(newEntries, i);
    }
    _entries = newEntries;
    _capacity = newCapacity;
}

private void NewBucketIndexes(Entry<TKey, TValue>[] newEntries, int i)
{
    uint hashCode = newEntries[i].hashCode;
    int bucketIndex = GetBucketIndex(hashCode);
    newEntries[i].next = _buckets[bucketIndex] - 1;
    _buckets[bucketIndex] = i + 1;
}

private int GetBucketIndex(uint hashCode) =>
    (int)(hashCode % _buckets.Length);

private int GetNextPrime(int currentCapacity)
{
    bool IsPrime(int number)

```

```

{
    if (number <= 1) return false;
    if (number == 2 || number == 3) return true;
    if (number % 2 == 0) return false;

    int sqrt = (int)Math.Sqrt(number);
    for (int i = 3; i <= sqrt; i += 2)
    {
        if (number % i == 0)
            return false;
    }
    return true;
}

```

```

int newCapacity = currentCapacity * 2;
while (true)
{
    if (IsPrime(newCapacity))
        return newCapacity;
    newCapacity++;
}

```

```

private ICollection<T> GetCollection<T>(Func<Entry<TKey, TValue>, T> selector)
{
    List<T> collection = new List<T>(_count);
    foreach (var entry in _entries)
    {
        if (entry != null)
        {
            T selectedValue = selector(entry);
            collection.Add(selectedValue);
        }
    }
}

```

```

        return collection;
    }

    private Entry<TKey, TValue>? FindEntryOfKey(TKey key)
    {
        uint hashCode = (uint)key.GetHashCode();
        int bucketIndex = GetBucketIndex(hashCode);
        int entryIndex = _buckets[bucketIndex] - 1;
        int next = -2;

        if (entryIndex <= -1)
            return null;

        do
        {
            Entry<TKey, TValue> entry = _entries[entryIndex];
            if (entry is null)
                return null;

            next = entry.next;

            if (hashCode == entry.hashCode)
                return entry;
            entryIndex = next;
        } while (next != -1);

        return null;
    }

    private class MyEnumerator : IEnumerator<KeyValuePair<TKey, TValue>>
    {
        private readonly MyDictionary<TKey, TValue> _dictionary;
        private int _pointer;
    }

```

```

public MyEnumerator(MyDictionary<TKey, TValue> dictionary)
{
    _dictionary = dictionary;
    _pointer = -1;
}

public KeyValuePair<TKey, TValue> Current
{
    get
    {
        if (_pointer >= 0 && _pointer < _dictionary._entries.Length)
        {
            var entry = _dictionary._entries[_pointer];
            if (entry != null)
                return new KeyValuePair<TKey, TValue>(entry.key, entry.value!);
        }
        throw new InvalidOperationException();
    }
}

```

```

object IEnumerator.Current => Current;

```

```

public bool MoveNext()
{
    _pointer++;

    while (_pointer < _dictionary._entries.Length)
    {
        var entry = _dictionary._entries[_pointer];
        if (entry != null)
        {
            return true;
        }
    }
}

```

```

        _pointer++;
    }

    return false;
}

public void Reset()
{
    _pointer = -1;
}

public void Dispose()
{
}
}

```

## Program.cs:

```

using MyDictionary;

var myDictionary = new MyDictionary<int, string>();

myDictionary.AddedPair += (sender, args) =>
    Console.WriteLine($"Pair [{args.key}, {args.value}] has been added");
myDictionary.RemovedPair += (sender, args) =>
    Console.WriteLine($"Pair [{args.key}, {args.value}] has been removed");
myDictionary.ChangedValue += (sender, args) =>
    Console.WriteLine($"Value by key [{args.key}] has been set to [{args.value}]");
myDictionary.Cleared += (sender, args) =>
    Console.WriteLine($"Dictionary has been cleared");

myDictionary.Print();
myDictionary.Add(1, "one");
myDictionary.Add(new KeyValuePair<int, string>(2, "two"));
myDictionary.Add(4, "!five!");
myDictionary[4] = "four";
myDictionary.Print();

myDictionary.Remove(1);
//myDictionary.Remove(9);
myDictionary.Remove(new KeyValuePair<int, string>(2, "two"));
myDictionary.Print();

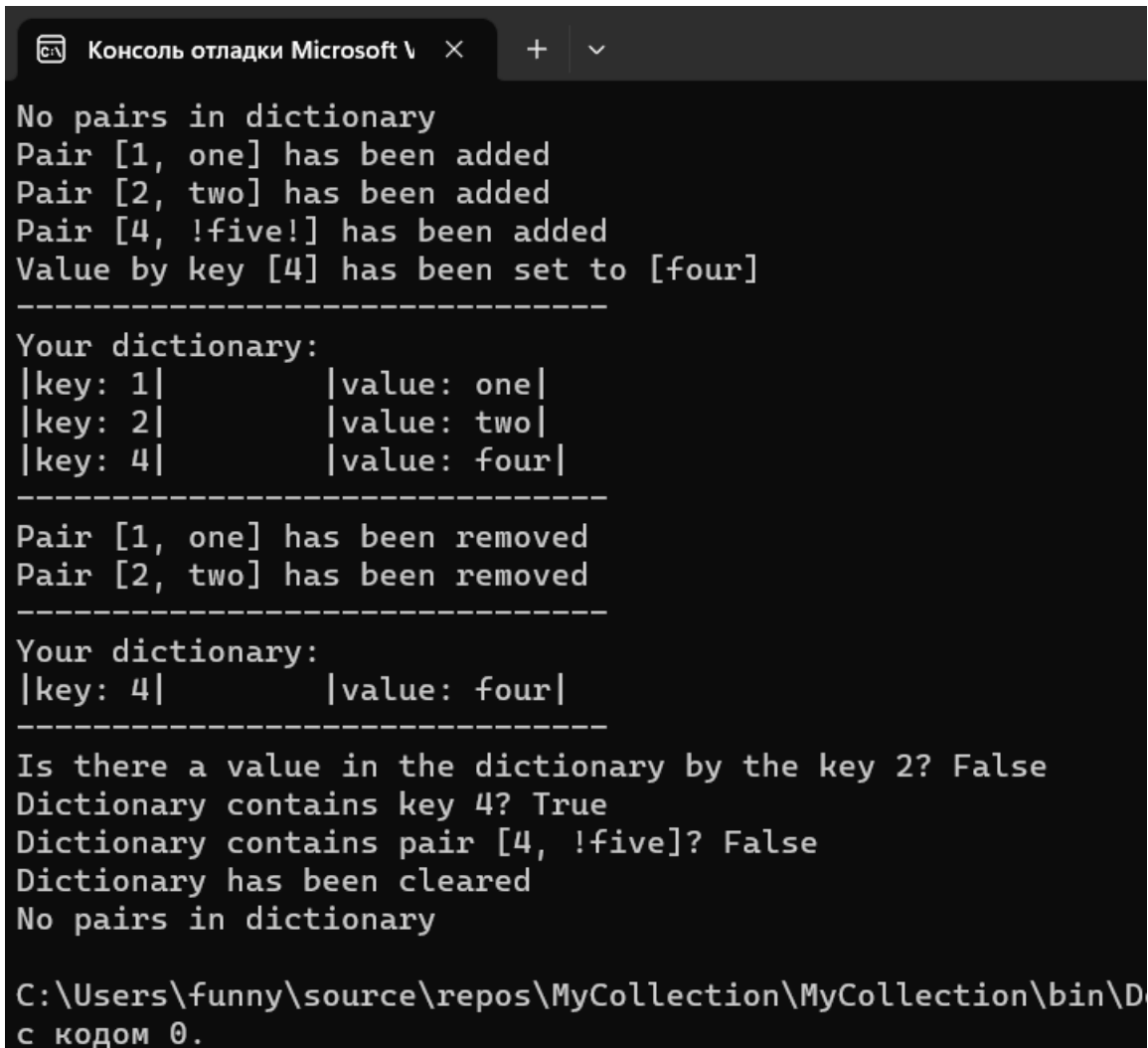
Console.WriteLine($"Is there a value in the dictionary by the key 2? " +
    $"{myDictionary.TryGetValue(2, out string? two)}");
Console.WriteLine($"Dictionary contains key 4? " +
    $"{myDictionary.ContainsKey(4)}");
Console.WriteLine($"Dictionary contains pair [4, !five]? " +
    $"{myDictionary.Contains(new KeyValuePair<int, string>(4, "!five"))}");

myDictionary.Clear();

```

```
myDictionary.Print();
```

Результати виконання:



```
Консоль отладки Microsoft V  X + v
No pairs in dictionary
Pair [1, one] has been added
Pair [2, two] has been added
Pair [4, !five!] has been added
Value by key [4] has been set to [four]
-----
Your dictionary:
|key: 1|          |value: one|
|key: 2|          |value: two|
|key: 4|          |value: four|
-----
Pair [1, one] has been removed
Pair [2, two] has been removed
-----
Your dictionary:
|key: 4|          |value: four|
-----
Is there a value in the dictionary by the key 2? False
Dictionary contains key 4? True
Dictionary contains pair [4, !five]? False
Dictionary has been cleared
No pairs in dictionary

C:\Users\funny\source\repos\MyCollection\MyCollection\bin\D
с кодом 0.
```

**Висновки:** в процесі виконання першої лабораторної роботи я навчився проектувати та реалізовувати узагальнені типи, а також типи з підтримкою подій.