Національний технічний університет України «КПІ»

Факультет інформатики та обчислювальної техніки

Кафедра Інформаційних систем та технологій

Лабораторна робота №1

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

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

Виконав:

студент гр. ІС-11

Воробйов Олексій

Викладач:

Бардін В.

2023 рік

**Мета**: 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 10:**



**Посилання на код GitHub:**

<https://github.com/Limbo2332/Lab1-GenericCollections>  
Код бібліотеки:

**MyLinkedListNode:**

namespace GenericCollection.Collections

{

/// <summary>

/// Element of linked list

/// </summary>

/// <typeparam name="T">Type of linked list</typeparam>

public class MyLinkedListNode<T>

{

public MyLinkedListNode(T value)

{

Value = value;

}

public T Value { get; internal set; }

public MyLinkedListNode<T>? Previous { get; internal set; }

public MyLinkedListNode<T>? Next { get; internal set; }

}

}

**MyLinkedList:**

using System.Collections;

namespace GenericCollection.Collections

{

/// <summary>

/// Linked list

/// </summary>

/// <typeparam name="T">Type of linked list</typeparam>

public class MyLinkedList<T> : ICollection<T>, IDisposable

{

#region Error messages constants

private const string NO\_SPACE\_IN\_ARRAY = "Array doesn't have enough space to copy";

#endregion

#region Collection events

/// <summary>

/// Event invokes when new item added

/// </summary>

public event Action<T>? ItemAdded;

/// <summary>

/// Event invokes when item removed

/// </summary>

public event Action<T>? ItemRemoved;

/// <summary>

/// Event invokes when collection has been cleared

/// </summary>

public event Action? CollectionCleared;

#endregion

#region First and last elements properties

private MyLinkedListNode<T>? \_firstElement;

private MyLinkedListNode<T>? \_lastElement;

/// <summary>

/// Get first element of collection

/// </summary>

public MyLinkedListNode<T>? First => \_firstElement;

/// <summary>

/// Get last element of collection

/// </summary>

public MyLinkedListNode<T>? Last => \_lastElement;

#endregion

#region Count property

/// <summary>

/// Get amount of nodes in collection

/// </summary>

public int Count { get; private set; }

/// <summary>

/// Checks if collection is read only

/// </summary>

public bool IsReadOnly => false;

#endregion

#region Constructors

public MyLinkedList() { }

public MyLinkedList(IEnumerable<T> collection)

{

if (!collection.Any() || collection is null)

{

throw new ArgumentNullException(nameof(collection));

}

foreach (T item in collection)

{

AddLast(item);

}

}

#endregion

#region IEnumerable implementation

public IEnumerator<T> GetEnumerator()

{

MyLinkedListNode<T>? current = \_firstElement;

while (current is not null)

{

yield return current.Value;

current = current.Next;

}

}

IEnumerator IEnumerable.GetEnumerator()

{

return ((IEnumerable<T>)this).GetEnumerator();

}

#endregion

#region Adding node

/// <summary>

/// Add element to the linked list

/// </summary>

/// <param name="value">Element value to add</param>

public void Add(T value)

{

MyLinkedListNode<T> newNodeElement = new MyLinkedListNode<T>(value);

Add(newNodeElement);

}

/// <summary>

/// Add element to the linked list

/// </summary>

/// <param name="node">Node to add</param>

public void Add(MyLinkedListNode<T> node)

{

AddLast(node);

}

/// <summary>

/// Add element to the start of linked list

/// </summary>

/// <param name="value">New first element value to add</param>

public void AddFirst(T value)

{

MyLinkedListNode<T> newFirstElement = new MyLinkedListNode<T>(value);

AddFirst(newFirstElement);

}

/// <summary>

/// Add element to the start of linked list

/// </summary>

/// <param name="node">New first node to add</param>

public void AddFirst(MyLinkedListNode<T> node)

{

MyLinkedListNode<T>? oldFirstElement = \_firstElement;

\_firstElement = node;

\_firstElement.Next = oldFirstElement;

if (Count == 0)

{

\_lastElement = \_firstElement;

}

if (oldFirstElement is not null)

{

oldFirstElement.Previous = \_firstElement;

}

Count++;

ItemAdded?.Invoke(node.Value);

}

/// <summary>

/// Add element to the end of linked list

/// </summary>

/// <param name="value">New last element value to add</param>

public void AddLast(T value)

{

MyLinkedListNode<T> newLastElement = new MyLinkedListNode<T>(value);

AddLast(newLastElement);

}

/// <summary>

/// Add element to the end of linked list

/// </summary>

/// <param name="value">New last node to add</param>

public void AddLast(MyLinkedListNode<T> node)

{

MyLinkedListNode<T>? oldLastElement = \_lastElement;

\_lastElement = node;

\_lastElement.Previous = oldLastElement;

if (Count == 0)

{

\_firstElement = \_lastElement;

}

if (oldLastElement is not null)

{

oldLastElement.Next = \_lastElement;

}

Count++;

ItemAdded?.Invoke(node.Value);

}

#endregion

#region Removing node

/// <summary>

/// Remove element from collection by value

/// </summary>

/// <param name="value">Value of node to remove</param>

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

public bool Remove(T value)

{

if (!Contains(value))

{

return false;

}

MyLinkedListNode<T> nodeToRemove = Find(value)!;

return Remove(nodeToRemove);

}

/// <summary>

/// Remove element from collection

/// </summary>

/// <param name="node">Node to remove</param>

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

public bool Remove(MyLinkedListNode<T> node)

{

if (Count == 0 || Count == 1 || node.Next is null)

{

return RemoveLast();

}

if (node.Previous is null)

{

return RemoveFirst();

}

if (!Contains(node.Value))

{

return false;

}

ItemRemoved?.Invoke(node.Value);

node.Next.Previous = node.Previous;

node.Previous.Next = node.Next;

Count--;

return true;

}

/// <summary>

/// Remove first element of collection

/// </summary>

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

public bool RemoveFirst()

{

if (Count == 0)

{

return false;

}

ItemRemoved?.Invoke(\_firstElement!.Value);

if (Count == 1)

{

\_firstElement = \_lastElement = null;

}

else

{

\_firstElement = \_firstElement!.Next;

\_firstElement!.Previous = null;

}

Count--;

return true;

}

/// <summary>

/// Remove last element of collection

/// </summary>

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

public bool RemoveLast()

{

if (Count == 0)

{

return false;

}

ItemRemoved?.Invoke(\_lastElement!.Value);

if (Count == 1)

{

\_lastElement = \_firstElement = null;

}

else

{

\_lastElement = \_lastElement!.Previous;

\_lastElement!.Next = null;

}

Count--;

return true;

}

#endregion

#region Clearing collection

/// <summary>

/// Clear collection

/// </summary>

public void Clear()

{

MyLinkedListNode<T>? current = \_firstElement;

while (current is not null)

{

current = current.Next;

}

\_firstElement = null;

Count = 0;

CollectionCleared?.Invoke();

}

#endregion

#region Finding node

/// <summary>

/// Find node by value

/// </summary>

/// <param name="value">Value of node to find</param>

/// <returns>Node to find or null if collection doesn't contain it</returns>

public MyLinkedListNode<T>? Find(T value)

{

MyLinkedListNode<T>? current = \_firstElement;

while (current is not null)

{

if (current.Value!.Equals(value))

{

return current;

}

current = current.Next;

}

return null;

}

/// <summary>

/// Find node by index

/// </summary>

/// <param name="index">Number of item in collection</param>

/// <returns>Node to find or null if collection doesn't contain it</returns>

public MyLinkedListNode<T>? Find(int index)

{

if (index < 0 || index > Count)

{

throw new ArgumentOutOfRangeException(nameof(index));

}

int count = 0;

MyLinkedListNode<T>? current = \_firstElement;

if (current is null)

{

return null;

}

while (count < index)

{

if (current is null)

{

return null;

}

count++;

current = current.Next;

}

return current;

}

/// <summary>

/// Check if element is in collection

/// </summary>

/// <param name="value">Value of node to find</param>

/// <returns>true if element is in collection, false if not</returns>

public bool Contains(T value)

{

return Find(value) is not null;

}

#endregion

#region Coping

/// <summary>

/// Copies elements from collection to array

/// </summary>

/// <param name="array">Array to copy to</param>

/// <param name="arrayIndex">To start copy from</param>

/// <exception cref="ArgumentNullException"></exception>

/// <exception cref="ArgumentOutOfRangeException"></exception>

/// <exception cref="ArgumentException"></exception>

public void CopyTo(T[] array, int arrayIndex)

{

if (array is null)

{

throw new ArgumentNullException(nameof(array));

}

if (arrayIndex < 0 || arrayIndex > array.Length)

{

throw new ArgumentOutOfRangeException(nameof(arrayIndex));

}

if (array.Length - arrayIndex < Count)

{

throw new ArgumentException(NO\_SPACE\_IN\_ARRAY);

}

MyLinkedListNode<T>? current = \_firstElement;

while (current is not null)

{

array[arrayIndex++] = current.Value;

current = current.Next;

}

}

#endregion

#region Indexer

/// <summary>

/// Get item by index

/// </summary>

/// <param name="index">Number of item in collection</param>

/// <returns>Node to find or null if collection doesn't contain it</returns>

public MyLinkedListNode<T>? this[int index]

{

get => Find(index);

}

#endregion

#region Disposable

public void Dispose()

{

Clear();

ItemAdded = null;

ItemRemoved = null;

CollectionCleared = null;

}

#endregion

}

}

**Висновки:** в ході роботи я навчився проектувати та реалізовувати узагальнені типи, а також типи з підтримкою подій.

**Питання до роботи:**

*1)* *Дайте визначання колекції. Як колекція пов’язана з інтерфейсами*

*IEnumerable та ICollection?*

Колекція – це набір зв’язаних, однотипних об’єктів динамічного розміру. Всі колекції повинні реалізувати інтерфейс ICollection, який в свою чергу реалізує інтерфейс IEnumerable.

*2)* *Розкажіть про основні інтерфейси необхідні для функціонування*

*колекцій та LINQ.*

Перш за все, колекція має імплементувати IEnumerable або IEnumerable<T> (якщо колекція узагальнена) інтерфейс. В інтерфейсі є лише один метод – GetEnumerator, який дозволяє ітерувати елементи колекції.

Для функціонування LINQ колекція повинна реалізовувати IQueryable або IQueryable<T> (якщо колекція узагальнеа) інтерфейс.

*3)* *Розкажіть про призначення і можливі сценарії застосування*

*інтерфейсу IEnumerator.*

Інтерфейс IEnumerator потрібен для того, щоб колекцію можна було ітерувати без прямого доступу до неї. Це потрібно, коли ми хочемо задати власну логіку перебору елементів в колекції. Інтерфейс містить метод MoveNext, який відповідає за перехід до наступного елементу, властивість Current для поточного елементу та метод Reset для того, щоб повернутись на початок.

*4)* *Порівняйте інтерфейси IEnumerable та IAsyncEnumerable.*

IEnumerable виконується синхронно, IAsyncEnumerable – асинхронно.

*5)* *Розкажіть про призначення і обмеження generic типів.*

Generic методи були створені для того, щоб програмісти могли працювати з різними типами даних, при цьому не дублюючи код. Ми створюємо загальне рішення, яке потім можемо використовувати з різними типами без необхідності створювати окремий код для кожного різного типу.

Обмеження для generic типів були додані для того, щоб звузити об’єм типів для узагальненого методу, класу, тощо.

*6)* *Поясніть призначення оператору default та його обмеження.*

Опетор default відповідає за те, щоб повернути значення за замовчуванням залежно від типу. Наприклад, int – 0, string – null, тощо.

Для того, щоб використовувати default, тип повинен мати значення за замовчуванням, тобто клас, інтерфейс не зможуть бути використані для цього. (Якщо вони не позначені як Nullable)

*7)* *Розкажіть про лямбда вирази. Наведіть приклад використання*

*лямбда-виразу.*

Лямбда-вираз – це скорочений запис анонімних функцій або такі делегати, де нам не потрібно визначати окремий метод. Приклад лямбда виразу:   
x => x+1

*8)* *Розкрийте різниці між expression & statement лямбдами.*

Перш за все, вони відрізняються виразами, які виконують та як повертають значення. Expression лямбди завжди повертають одне значення, коли statement взагалі не повертає значення. Прикладом Expression лямбди може бути делегат Func, тоді, коли для statement – Action.

*9)* *Розкажіть про події, і як вони реалізовані в С#.*

Подія – це механізм, при якому всі об’єкти можуть підписатись, отримати сповіщення та зреагувати на нього. Події реалізують патерн Observer. Визначення події базується на основі делегату.

*10)* *Поясніть, яким чином виконується підписання на події та скасування*

*підписки.*

Для підписки чи скасування події потрібен екземпляр делегату, який буде відповідати сигнатурі події.

Підписка на подію відбувається за допомогою оператора +=, коли скасування підписки за допомогою -=.

*11) Наведіть склад класу делегату та поясніть, чим забезпечується*

*контроль типів в делегатах.*

Делегат – це тип, який представляє собою посилання на метод. Вони зберігають та викликають методи. Контроль типів забезпечується сигнатурою делегату, тому сигнатура методу має відповідати сигнатурі делегату.